NPS52-81-012

NAVAL POSTGRADUATE SCHOOL

Monterey, California

ELEMENTS OF PROGRAMMING LINGUISTICS

Part I

The Lambda Calculus and its Implementation

B. J. MacLennan
Naval Postgraduate School

Approved for public release; distribution unlimited

Prepared for:
Naval Postgraduate School
Monterey, California 93940

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral J. J. Ekelund
Superintendent

D. A. Schrady
Acting Provost

The work reported herein was supported by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research.

This report was prepared by:

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NPS52-81-012	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) ELEMENTS OF PROGRAMMING LINGUISTICS Part I The Lambda Calculus and its Implementation		5. TYPE OF REPORT & PERIOD COVERED Technical Report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Bruce J. MacLennan		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61152N; rr000-01-10 N0001481WR10034
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93940		12. REPORT DATE 13 August 1981 13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

lambda calculus, LISP, block structured languages, list processing, interpreters, stack implementation, reduction, translation

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The lambda calculus is used as an introduction to programming language concepts, particularly the concepts of functional programming. Both interpreted and compiled implementations of an extended lambda calculus are discussed. They can be adopted to implementations of Pascal and Lisp it is shown that traditional stack-based run-time structures can be directly derived from the reduction rules of the lambda calculus.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

Unclassifed

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

TABLE OF CONTENTS

3. primitives

4. data types

5. the list data type

6. recursive problem solving

7. syntactic sugar

8. local declarations in mathematical prose

9. local declarations in the lambda calculus

10. terminology

11. compound declarations

12. recursive declarations

4. Implementing the Lambda Calculus

This chapter develops a recursive interpreter for the lambda calculus. It is similar to the traditional EVAL interpreter for LISP. The structure of this interpreter is motivated by deriving it in a series of simple steps from the hand-reduction procedures presented in Chapter 2. The important concepts of closures and association lists are introduced. This chapter finishes by discussing the elimination of the run-time lookup of variables. This leads to the ideas of (ip,ep) pairs, static nesting levels and static distances. Contour diagrams are used to clarify these concepts. This material prepares the ground for the discussion of static chains in Chapter 5.

1. goals defined

2. mechanical reduction

3. context

4. hand evaluation

5. constants

6. variables

7. abstractions

8. applications

9. primitive applications

10. conditionals

11. example of hand evaluation

12. representation defined

13. field accessing functions

14. association lists

15. closures

16. mechanical evaluation

17. representation of variables by fixed locations

18. static nesting level

19. multiple parameters

5. Runtime Organization

Chapter 5 develops an implementation of the extended lambda calculus for a conventional computer. As preliminaries the ideas of stacks and postfix instructions are explained. This simplifies the explanation of the L-machine, a simplified postfix-oriented stack computer. This material provides the background for a first cut at a compiled implementation of the extended lambda calculus. The ideas of activation records and static chains are explained and derived from the simple interpreter described in Chapter 4. This allows the development of L-code sequences for each ELC (extended lambda calculus) construct.

1. goals defined

2. stacks

3. postfix instructions

4. the L-machine architecture

5. activation record structure

6. translation to L-code
    1. constants
    2. primitive applications
    3. variables
    4. applications
    5. abstractions
    6. conditionals
    7. blocks

Chapter 1

I.1 Introduction

There are several reasons that the lambda calculus is important for designers and implementors of programming languages. One of the first uses of the lambda calculus as a tool of programming linguistics occurred in the mid 1960s when Strachey and Landin used it to elucidate the semantics of Algol-60. The lambda calculus was picked because, as we shall see, it has the same scope rules as block structured languages. This allowed the consequences of these rules to be studied in a distilled and simple context. In particular, this allowed the implementation of block structured languages to be investigated without the complexities of "real" languages. In this manual you will see both interpreted and compiled implementations of the lambda calculus - seminal techniques that can be used or adapted for many other languages. For instance, the closure implementation concept is important in understanding coroutines, processes and the "lazy evaluation" techniques discussed in Part II.

The symmetry and simplicity of the lambda calculus sets a standard against which all languages can be compared. For instance, the essential equivalence between formal parameters and declarations in the lambda calculus suggests a solution to many problems of language design. This has been used in several experimental languages, e.g. Quest.

Although the lambda calculus is capable of computing any computable function, it provides a model of computation that is much closer to real languages than most other such models (e.g. Markov algorithms, Turing machines, recursive functions). It is therefore more relevent to the real problems of language design.

The programming practices and modes of thought used with the lambda calculus and its derivatives (such as LISP) have formed the foundation of functional programming - the method of programming recently popularized by Backus and characterized by a high-level applicative programming style. The lambda calculus is essential for an understanding of the design and implementation of such languages.

CHAPTER 2

I.2 The Lambda Calculus

I.2.1 calculus defined.

We will be using the lambda calculus as a model of computation. That is, we will be using the lambda calculus as a way of describing the meaning of programs and as a guide to the implementation of programming languages. A calculus is a notation which can be manipulated mechanically to achieve some end. "Calculus" is the Latin word for "pebble" and is derived from the fact that people used to do arithmetic by manipulating pebbles. ("Calc" is the Latin word for "limestone" and is also the basis for words such as "calculate".) You are probably familiar with the differential and integral calculi. In these it is possible to manipulate formulas according to the rules of integration and differentiation to get results that would otherwise have to be derived by solving complicated limits. You may also be familiar with the propositional and predicate calculi. In these certain forms of deductive reasoning can be performed by the mechanical manipulation of symbols.

The reason for developing a calculus is that by reducing some process to a set of simple mechanical rules one decreases the chances of making an error. Of course, the fact that the rules of a calculus are mechanical and strictly defined makes them ideal for manipulation by computer. The lambda calculus is a calculus which models (mimics) the process of computation itself. Under the now generally accepted definition of computability, it has been shown (by Alonzo Church, the inventor of the lambda calculus, and Alan Turing) that anything that can be done on any computer can also be done in the lambda calculus (although perhaps very inefficiently).

I.2.2 bound variables defined.

One of the central ideas of the lambda calculus is that of a bound variable (sometimes called a dummy variable). Bound variables are common in all mathematical notations, for instance, in the summation

$$\sum_{i=1}^{n} A_i$$

the 'i' is the bound variable. It is a characteristic of bound

variables that it doesn't matter what they are. For instance,

$$\sum_{k=1}^{n} A_k$$

means exactly the same thing as the previous summation. Similarly, the integral of x^2-3x with respect to x:

$$\int_0^t x^2-3x\ dx$$

is the same as the integral of u^2-3u with respect to u:

$$\int_0^t u^2-3u\ du$$

In set theory, the set of all x such that $x \geq 0$ is the same as the set of all n such that $n \geq 0$:

$$\{x | x \geq 0\} = \{n | n \geq 0\}$$

Also, a proposition such as "for every x, $x+1>x$":

$$\forall x\{\ x+1>x\ \}$$

is the same as the proposition "for every y, $y+1>y$":

$$\forall y\{\ y+1>y\ \}$$

In the following examples the bound variables are listed on the right.

Expression	Bound Variable	Verbalization
$\sum_{i=1}^{n} A_i$	i	the sum for i from 1 ...
$\prod_{j=1}^{n} f(j)$	j	the product for j from 1 ...
$d(x^2-3x)/dx$	x	the derivative with respect to x of
$D_y(y^2-3y)$	y	the derivative with respect to y of
$\int_0^x x^2-3x dx$	x	the integral with respect to x of
$\{ x \mid x>0 \}$	x	the set of all x such that
$\forall x\{ x+1>x \}$	x	for all x,
$\exists y\{ 2y=y \}$	y	there exists a y such that
$\varepsilon x.x>y$	x	any x such that
$\iota y.2y=1$	y	the unique y such that

I.2.3 bound and free occurrences, and scope.

Two ideas that will be very useful to us are bound occurrence and free occurrence. Consider the expression

$$i \sum_{j=1}^{n} A_{ij}$$

The occurrence of 'j' in 'A_{ij}' is called a bound occurrence of the variable 'j'. It is bound by the summation operator (Σ), which is called the binding site (or just binding) of this occurrence of 'j'. We can see that 'j' is bound by noting that we can change it to any other variable (except 'i') without changing the meaning of the expression. For instance,

$$i \sum_{k=1}^{n} A_{ik}$$

Any occurrence of a variable that is not a bound occurrence is called a free occurrence. For instance, 'i', 'n' and 'A' all occur free in the above expression. Clearly, if we change a free variable we have changed the meaning of the expression:

$$i \sum_{j=1}^{n} M_{ij}$$

Notice that when we say that an occurrence of a variable is bound or free, we say this relative to some expression. For instance, 'j' is free in

$$A_{ij}$$

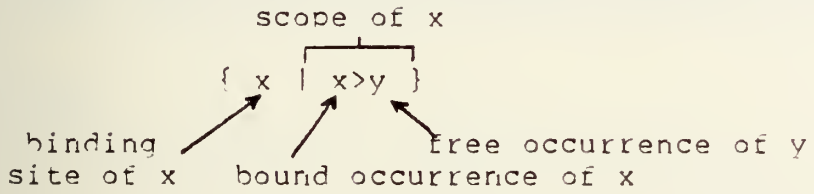
but is bound in

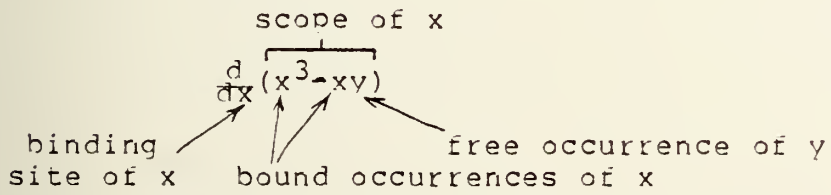
$$\sum_{j=1}^{n} A_{ij} \quad \text{and} \quad i! \sum_{j=1}^{n} A_{ij}$$

Similarly, 'i' is free in the above expressions, but bound in

$$\sum_{i=1}^{m} (i! \sum_{j=1}^{n} A_{ij})$$

The binding site of a variable determines its scope, which is the region of the expression over which that variable is bound. This region is usually indicated by some textual convention, such as brackets or parentheses. To state things differently, all occurrences of a variable which are in the scope of a binding of that variable are bound occurrences of that variable. The following figures exemplify these concepts:





As we have already seen, it is perfectly meaningful for scopes to be nested within other scopes. Some examples of nested scopes are shown below (the brackets indicating scope are called scoping lines):

$$\sum_{i=1}^{m} (i \sum_{j=1}^{n} A_{ij})$$

scope of j

scope of i

$$\int_0^1 x \int_0^x y^2+xy \, dy \, dx$$

scope of y

scope of x

We can summarize these ideas as follows:

- The binding site of a variable determines its scope.
- An occurrence of a variable is bound if it is in the scope of a binding site of that variable.
- An occurrence of a variable is free otherwise.

EXERCISES:

For each variable occurrence in the following expressions, indicate whether it is a binding site, a bound occurrence, or a free occurence. Draw scoping lines to indicate the scope of each binding.

1. $\{ n \mid n>m \}$
2. $\int_0^1 x \sin(x/y) \, dx$
3. $\int_0^1 x \int_0^x \sin(yx) dy \, dx$
4. $\frac{\partial}{\partial x}\frac{\partial}{\partial y}\left[\frac{x^2+y^2}{xy}\right]$
5. $\sum_{i=1}^{n}\sum_{j=1}^{i} A_{ij}B_{ji}$
6. $\forall x[x \in Z \Rightarrow \exists y(y \in Z \wedge x=y+1)]$
7. $\{x|x>0\} \cup \{x|x<0\} = \{x|x\neq 0\}$
8. $\sinh(x) = \frac{e^x-e^{-x}}{2}$

1.2.4 renaming bound variables.

As we have seen, bound variables are arbitrary. This is one reason they are often called dummy variables; they only serve to establish a connection between parts of an expression. Bound variables are the pronouns of mathematics.

Changing a bound variable to another variable does not change the meaning of an expression. For instance,

$$\sum_{i=1}^{m} A_{ij} \quad \text{and} \quad \sum_{k=1}^{m} A_{kj}$$

both mean the sum of the j-th column of the matrix A. Suppose we change the bound variable to 'j':

$$\sum_{j=1}^{m} A_{jj}$$

This sums the diagonal of the matrix; we have altered the meaning of the expression! We got into this trouble because we changed the bound variable 'i' to the variable 'j', which already occurred within the expression. Thus, the occurrence of 'j' in A_{jj} became accidently bound. This is called a collision of variables. The conclusion that can be drawn from this is: we can change a bound variable, throughout its scope, to another variable only if the latter variable does not occur within that scope.

EXERCISES:

For each expression determine whether the indicated change of variable alters the meaning of the expression.

1. $\{x|x>y\}$; change x => z.

2. $\{x|x>y\}$; change x => y.

3. $\frac{d}{dx}(x^3-xy)$; change x => t.

4. same; change x => y.

5. $\forall x[\exists y(y>x)]$; y => x.

6. $\frac{\partial}{\partial x}\frac{\partial}{\partial y}\left[\frac{x^2+y^2}{xy}\right]$; y => x.

7. $\sinh(x) = \frac{e^x-e^{-x}}{2}$; x => u.

8. $\{m|m>e\} \cup \{e|\ \sin(e)=0\ \}$; $e \Rightarrow m$.

9. $\{x|x>0\} \cup \{y|v<0\}$; $v \Rightarrow x$.

10. $\sum_{i=1}^{m} A_i + \sum_{j=1}^{n} B_j$; $i \Rightarrow j$.

11. $\sum_{i=1}^{m} i \cdot \sum_{j=1}^{n} V_j$; $j \Rightarrow i$.

I.2.5 function definition.

When bound variables are used in function definitions they are often called formal parameters. For example, in

$$f(x) = x^2-3x$$

'x' is the bound variable or formal parameter. This is different from the previous examples of bound variables in that the bound variable and the function name are tied together. The lambda calculus provides a notation for functions which does not have this problem. For instance, in the lambda calculus the function f can be written

$$\lambda x\{x^2-3x\}$$

(the λ is a lambda), which can be read "that function which takes any x into x^2-3x." In the lambda calculus, if we have defined

$$f = \lambda x\{x^2-3x\}$$

and then we ask the value of 'f(5)', we can find it by substituting '5' for 'x' throughout its scope. More specifically, we start with

$$f(5)$$

When we substitute $\lambda x\{x^2-3x\}$ for 'f' we get

$$\lambda x\{x^2-3x\}(5)$$

Now, we replace this expression by a copy of the body of f (namely x^2-3x) in which every free occurrence of 'x' is replaced by '5':

$$5^2-3\cdot5 = 25-15 = 10$$

This is called the "copy rule" for function evaluation because the invocation 'f(5)' is actually replaced by a copy of the body of the function with its parameter textually substituted, i.e.

'$5^2-3\cdot5$'.

I.2.6 syntax of the lambda calculus.

The lambda calculus has a very simple syntax. Lambda expressions are composed of the symbols 'λ', '{', '}', '(', ')' and variable names, put together according to the following rules:

(1) If 'x' is a variable and 'E' is an expression of the lambda calculus, then 'λx{E}' is an expression of the lambda calculus, called an abstraction. We call 'x' the binding of the abstraction and 'E' the body of the abstraction.

(2) If 'F' and 'E' are expressions of the lambda calculus, then 'F(E)' is an expression of the lambda calculus, called an application. We call 'F' the operator of the application and 'E' the operand of the application.

(3) If 'E' is an expression of the lambda calculus, then so is '(E)'.

(4) If 'x' is a variable, then it is an expression of the lambda calculus.

(5) The only lambda expressions are those described in (1) to (4).

To permit more meaningful examples, we will sometimes allow additional types of expressions within our lambda expressions, such as conventional arithmetic expressions (e.g. '3+x').

The way in which we have described the syntax of the lambda calculus will form a model for all later syntax descriptions. We have enumerated a set of primitives (the basic symbols and variables) and have described a set of constructors or formation rules, which will yield all the legal expressions of the language when applied recursively to the primitives.

EXERCISES:

Which of the following are legal lambda calculus expressions?

1. x

2. f(x)

3. λx{f(x)}

4. (g)y

5. f(a)(b)

6. λx{f(x)} (a)

7. λx{r(x)}(λx{x})

8. f g

9. {g(a)}

10. f{x}

11. f(λx)

I.2.7 semantics of the lambda calculus.

In a previous section we informally discussed evaluation of expressions of the lambda calculus by the copy rule. In this section this evaluation is defined more exactly through two reduction rules:

1 (renaming): one expression may be reduced to another by changing a bound variable throughout its scope to any other variable that does not occur within that scope.

2 (substitution): a subexpression of the form 'λx{E}(A)' may be reduced by replacing it by a copy of E in which all free occurrences of x are replaced by A, provided this does not result in any free variables of A becoming bound.

We can restate the renaming rule as follows: An expression 'λx{E}' may be reduced by renaming to an expression 'λy{F}', where F is obtained from E by replacing all free occurrences of x in E by y. This is only allowed if y does not occur in E. For example, suppose we wish to rename x to u in 'λx{x^2+2x+1}'. We do this by changing to u all free occurrences of x in 'x^2+2x+1'. This yields 'λu{u^2-2u+1}'. This reduction is symbolized:

$$\lambda x\{x^2+2x+1\} \Rightarrow \lambda u\{u^2+2u+1\}$$

Some other reductions permitted by the renaming rule are:

λx{x} => λa{a} => λg{g} => λr{r}
λx{λy{x(y)}} => λr{λy{r(y)}} => λr{λa{r(a)}}
λx{x+1} => λu{u+1}
λx{λy{x+y}} => λa{λy{a+y}} => λa{λb{a+b}}

Lets consider an illegal application of the renaming rule in order to better understand its restriction. Suppose we wish to apply the renaming rule to

$$f = \lambda x\{ e \cdot x \} = \lambda x\{ (2.71828...)x \}$$

by changing x to e. This is not allowed since e occurs in 'e·x'. We can see that if we did the substitution anyway we would change the meaning of the expression:

$$f' = \lambda e\{ e \cdot e \} = \lambda e\{ e^2 \}$$

Note that f(1) = 2.71828... while f'(1) = 1; f and f' are not the same function. Renaming x to y, however, would not change the meaning:

$$f'' = \lambda y\{ e \cdot y \}; \quad f''(1) = 2.71828...$$

The renaming rule is generally needed only to avoid variable collisions.

EXERCISES:

Apply the renaming rule as indicated, or state that its application would be illegal:

1. $\lambda x\{x\}$; change x => y.

2. $\lambda x\{\lambda y\{x+y\}\}$; change y => x.

3. $\lambda x\{f(x)\}$; change f => g.

4. $\lambda d\{d+e\}$; change d => e.

5. $\lambda x\{\lambda y\{x(y)\}\}$; change x => f.

Next we will consider the substitution rule. The expression '$\lambda x\{x+1\}(3)$' fits the form required by the substitution rule: it is an application whose operator is an abstraction. Hence, we can reduce it by replacing all free occurrences of 'x' in 'x+1' by '3'. The result is '3+1'. Now consider

$$\lambda x\{ \lambda y\{x(y)\} \}(f)$$

This is an application whose operator is the abstraction

$$\lambda x\{ \lambda y\{x(y)\} \}$$

We can apply the substitution rule by replacing by 'f' all free occurrences of 'x' in '$\lambda y\{x(y)\}$'. This produces:

$$\lambda y\{f(y)\}$$

To better understand the restriction on the substitution rule, first consider this legal substitution: Suppose, as is usual, that e=2.71828.... Let f = λx{λy{x+y}}. Then we can reduce f(2e)(1) as follows:

```
f(2e)(1)  =>  λx{λy{x+y}}(2e)(1)
=>  λy{ 2e+y }(1)
=>  2e+1  =>  5.43...+1  =>  6.43...
```

Now lets look at a slightly different example:

f'(2e)(1) where f' = λd{λe{d+e}}

We see that f' is the same function as f; we have just renamed x and y to d and e. When we replace f' by its value we get

λd{λe{d+e}}(2e)(1)

which is an application whose operator is the abstraction:

λd{λe{d+e}}

To apply the substitution rule we must replace all free occurrences of 'e' in 'λe{d+e}' by '2e'. But, this is only allowed if it does not cause a free variable of '2e' to become bound. In this case a collision does occur, since 'e' is free in '2e' but not in 'λe{d+e}'. To see the reason for this restriction we will go ahead and perform the substitution. The result will be 'λe{2e+e}'. This has changed the meaning of the expression; a fact we can see by evaluating:

λe{2e+e}(1) => 2·1+1 => 3

Hence, f'(2e)(1) => 3

although we know the answer should be 6.43.... How do we avoid this situation? Since bound variables are arbitrary, we simply rename the offending <u>bound</u> variable. For instance, we can rename 'e' to 'c':

λd{λe{d+e}}(2e)(1) => λd{λc{d+c}}(2e)(1)

which lets us proceed with the reduction:

```
λd{ λc{d+c} }(2e)(1)  =>  λc{2e+c}(1)
=>  2e+1  =>  5.43...+1  =>  6.43...
```

In fact, this is the major use of the renaming rule.

<u>EXERCISES</u>:

Determine if the substitution rule is applicable to each of these expressions. If so, reduce the expression by the substitution rule, first applying the renaming rule, if necessary.

1. λx{x(y)}(f)

2. λx{ λy{x(y)} }(y)

3. f(3)

4. λx{ λy{x(y)} }(λz{y(z)})

5. λy{y·y}(3)

6. λf{ f(3)+f(4) }(g)

I.2.8 reduced form.

If and when an expression is reduced to the extent that the substitution rule can no longer be applied to it, it is said to be in reduced form. Intuitively, an expression is in reduced form when it is an answer (i.e. it is done computing). The following table shows examples of both unreduced and reduced expressions:

Not Reduced	Reduced
λx{x}(y)	y
λy{f(y)}(a)	f(a)
λx{ λy{x(v)} }(f)	λy{f(y)}
λx{x}(λx{x})	λx{x}
λx{x(y)}(λx{x(x)})	y(y)

In each case above, the expression on the right is the reduction of the expression on the left. Not all expressions have a reduced form. Consider the expression Y(Y), where Y = λx{x(x)}:

Y(Y) => λx{x(x)}(Y) => Y(Y) => ...

This is the lambda calculus equivalent of an infinite loop.

EXERCISES:

Decide if each of the following expressions is in reduced form. If not, then reduce it to reduced form.

1. λf{ f(3)+f(4) }(λy{y·y})

2. f(λx{x+x})

3. $\lambda x\{x<0\}(\ \lambda x\{x+1\}\ )$

4. $\lambda x\{x<0\}(\ \lambda x\{x+1\}(2)\ )$

I.2.9 multiple parameters and other abbreviations.

The lambda expressions we have defined have only one bound variable. We can get the effect of two bound variables by nesting the lambda expressions, for instance:

$\lambda x\{\lambda y\{x+y\}\}(3)(1)$
$\Rightarrow\ \lambda y\{3+y\}(1)$
$\Rightarrow\ 3+1$
$\Rightarrow\ 4$

Because such nested lambda expressions are so common, we allow the following abbreviations (using '=>' also as a sign of abbreviation):

$\lambda xy\{x+y\}\ \Rightarrow\ \lambda x\{\lambda y\{x+y\}\}$
$F(3,1)\ \Rightarrow\ F(3)(1)$

These abbreviations are the usual method of handling multi-argument functions in the lambda calculus. Of course, it is not normally necessary to think of these as abbreviations; we just do the multiple parameter substitutions directly, e.g.,

if $f = \lambda xy\{x+y\}$
then $f(3,1)$

$\Rightarrow\ \lambda xy\{x+y\}(3,1)$
$\Rightarrow\ 3+1\ \Rightarrow\ 4$

In exactly the same way we will allow substitutions involving any number of parameters, including none.

$\lambda abc\{ax^2+bx+c\}(9,6,1)\ \Rightarrow\ 9x^2+6x+1$

$\lambda\{m+1\}()\ \Rightarrow\ m+1$

As we have seen in some of the previous examples, lambda expressions can get quite large. In order to be able to program significant functions in the lambda calculus we will need a way of attaching names to lambda expressions. Therfore we will allow rewriting rules of the form:

plusp => $\lambda x\{x>0\}$
minusp => $\lambda x\{x<0\}$
succ => $\lambda x\{x+1\}$
square => $\lambda x\{x\cdot x\}$

Then we can write, for instance,

```
minusp(succ(2))
=> λx{x<0}(succ(2))
=> succ(2)<0
=> λx{x+1}(2)<0
=> 2+1<0
=> 3<0
=> false
```

I.2.10 the Church-Rosser property.

In the above reduction we reduced the outermost application first. We need not have done this, for instance,

```
minusp(succ(2))
=> minusp(λx{x+1}(2))
=> minusp(2+1)
=> minusp(3)
=> λx{x<0}(3)
=> 3<0
=> false
```

We see that this produces the same answer. It is a property of the pure lambda calculus (called the Church-Rosser property) that any reduction sequence that terminates will produce the same result. It is important to know when a language, or a part of a language, satisfies the Church-Rosser property since this means an optimizing compiler can alter the evaluation order without effecting the meaning of a program. Since some of the extensions of the lambda calculus that we will be investigating do not satisfy the Church-Rosser property, we will adopt a standard order for reduction. It is defined by the rule: don't reduce an application until its arguments are already in reduced form.

> STANDARD REDUCTION ORDER: An application is reduced (by substitution) only if its arguments are already in reduced form.

EXERCISE: Reduce to reduced form:

```
λf{succ(f(2)+f(3))}(λx{square(x)+2})
```

EXERCISE: Suppose the following definitions are given:

```
Zero  => λfc{c}
One   => λfc{f(c)}
Two   => λfc{f(f(c))}
Three  => λfc{f(f(f(c)))}
     ...
sum   => λMN{λfc{M(f,N(f,c))}}
```

then, reduce to reduced form 'sum(Two,One)'. What is this equal to? If you wonder about the motivation for these definitions, then try reducing 'Three(succ,0)' and 'Two(succ,3)'.

Chapter 3

I.3 The Extended Lambda Calculus.

I.3.1 Conditionals.

The lambda-calculus, as we have been using it so far, is not very useful; it is only possible to define functions that evaluate strictly in order: there is no decision making ability. Therefore we would like to define a function 'if' such that if(c,t,f) => t if c is true and if(c,t,f) => f if c is false. Hence, 'true' selects 't' from (t,f) and 'false' selects 'f' from (t,f). One way to do this is to define 'true' and 'false' so that true(t,f) => t and false(t,f) => f. Thus:

```
true  => λtf{t}
false => λtf{f}
```

Then we want if(c,t,f) => c(t,f), where c reduces to true or false, so

```
if => λbtf{b(t,f)}
```

To see how this works, suppose 'x=y' returns 'true' if x equals y and 'false' otherwise. Then

```
if( 2=0, 25, 37)
=> if( false, 25, 37)
=> λbtf{b(t,f)} ( false, 25, 37)
=> false( 25, 37)
=> λtf{f}( 25, 37)
=> 37
```

Next we will program the logical connectives: 'and', 'or' and 'not'. Not is the simplest since it just negates a truth value:

```
not(true)  => false
not(false) => true
```

That is, if x is true then not(x) is false, otherwise not(x) is true. This can be directly translated to the lambda calculus:

```
not => λx{ if( x, false, true ) }
```

The 'and' function is defined so that and(x,y) is true only if both x and y are true. This is summarized in the following truth

table:

AND	true	false
true	true	false
false	false	false

We can see that if x is true then and(x,y) has the same value as y, and that if x is false then and(x,y) is false regardless of the value of y. We can translate this directly into the lambda calculus:

$$\text{and} \Rightarrow \lambda xy\{ \text{if}(x, y, \text{false}) \}$$

EXERCISE: Define 'or' so that or(x,y) is true if and only if x or y or both are true. That is, 'or' must satisfy the truth table:

OR	true	false
true	true	true
false	true	false

Show that your definition works by reducing 'or(false,true)'.

I.3.2 Recursive Definitions.

Now that we have a conditional, we can define some more useful functions. The factorial function is defined so that

$$4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$$
$$3! = 3 \cdot 2 \cdot 1 = 6$$
$$0! = 1$$

or in general,

$$n! = n(n-1)(n-2)\ldots(2)(1)$$

The factorial can also be defined recursively as follows:

$$n! = \begin{cases} 1, & \text{if } n=0 \\ n(n-1)!, & \text{if } n>0 \end{cases}$$

Using the conditional it is now easy to define a function 'fac' such that fac(n) = n!.

$$\text{fac} \Rightarrow \lambda n\{ \text{if}(n=0, 1, n \cdot \text{fac}(n-1)) \}$$

Consider the computation of fac(2):

```
fac(2)
=>  λn{ if( n=0, 1, n*fac(n-1)) } (2)
=>  if( 2=0, 1, 2*fac(2-1))
=>  if( false, 1, 2*fac(1))
=>  2*fac(1)
=>  2 * λn{ if( n=0, 1, n*fac(n-1)) } (1)
=>  2 * if( 1=0, 1, 1*fac(1-1) )
=>  2 * 1 * fac(0)
=>  2 * 1 * λn{ if( n=0, 1, n*fac(n-1)) } (0)
=>  2 * 1 * if( 0=0, 1, 0*fac(0-1))
=>  2 * 1 * if( true, 1, 0*fac(-1))
=>  2 * 1 * 1
=>  2 * 1
=>  2
```

To keep the above reduction readable, most of the reductions associated with 'if' have not been shown. Notice that in the fourth to the last line (the last line with an 'if' in it) if we had decided to reduce the argument formula 0*fac(-1) we would have started a never-ending recursion. That is,

```
2 * 1 * if( 0=0, 1, 0*fac(0-1))
=>  2 * 1 * if( true, 1, 0*fac(-1))
=>  2 * 1 * if( true, 1, 0 * λn{ if( n=0, 1, n*fac(n-1))}(-1))
=>  2 * 1 * if( true, 1, 0 * if( -1=0, 1,
              -1 * λn{ if( n=0, 1, n*fac(n-1)) }(-2) ))
=>  2 * 1 * if( true, 1, 0 * if( false, 1,
              -1 * if( -2=0, 1, -2*fac(-2-1) )))
=>  ...
```

This process may never terminate! What the Church-Rosser property really says is that two different reductions of a formula give the same result provided they both terminate. For this reason we avoid evaluating any arguments of 'if' that we don't have to.

Since 'if' is such a common function, we will introduce a special notation for it. This is just an abbreviation, it really adds nothing to the lambda calculus. Such abbreviations are often called "syntactic sugar" (because a little "syntactic sugar" helps one to swallow the lambda calculus). For simple 'if's, such as if(b,t,e) we will write [b->t|e], which is read: "if b then t else e." For nested 'if's, such as if(b, t, if(c, u, e)) we will write

```
[ b -> t | c -> u | e ]
```

and so forth. This can be read "if b then t else if c then u else e". Using this notation the factorial function can be

written

$$fac \Rightarrow \lambda n\{ [n=0 \rightarrow 1 \mid n \cdot fac(n-1)] \}$$

I.3.3 Primitives.

We have seen that it is possible to define in the lambda calculus Boolean values (true and false), logical connectives, 'if' expressions, arithmetic (sum), and even numbers themselves. This should lend some credibility to the statement made earlier that anything that can be done on a computer can be done in the lambda calculus. For our purposes, there is not much point in carrying this exercise any further. In the future, any application-oriented functions that we might need (such as arithmetic) will be introduced as extensions to the lambda calculus. For instance, the arithmetic operations might be introduced by a set of rules like:

```
sum(1,1)  =>  2
sum(1,2)  =>  3
....
```

or in general

```
sum(m,n)  =>  m+n
```

This way we have an application independent language framework formed by the lambda abstraction and function application syntax, and a flexible set of application dependent primitive operations which we can define as the need occurs. Notice that again we are breaking the language down into primitives and constructors. In the next section we will build a list processing language by combining the constructors of the lambda calculus with a set of powerful list processing primitives. We will find that many apparantly different programming languages are really just sugared versions of the lambda calculus with some application-oriented primitives added. The following chart shows some primitives that might be included in languages intended for numerical, list processing, string processing and data processing applications.

language	constructors	primitives
	application independent	application dependent
numerical	$\lambda x\{E\}$, f(a)	integers, reals, +, -, x, /, ...
list processing	$\lambda x\{E\}$, f(a)	lists, atoms, first, rest, cons, ...
string processing	$\lambda x\{E\}$, f(a)	strings, characters, substr, concat, match, ...
data processing	$\lambda x\{E\}$, f(a)	files, records, move, read, write, ...

I.3.4 Data Types.

When we extend the lambda calculus with new primitives we will always do it by defining a set of data values and a set of primitive operations on those values. Any operations we wish to perform on the data values must be constructed from the primitive operations. The term data type is used to refer to a set of data values together with a set of primitive operations on those values. For instance, the data type integer is the set of integer data values:

$$\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots$$

together with the primitive operations on those values, e.g.,

$$+, -, \times, /, =, \neq, <, >, \leq, \geq$$

Any other operation, e.g. squaring, must be constructed from the given values and primitive operations:

$$\text{square} = \lambda n\{ nxn \}$$

Similarly, the Boolean data type is composed of the Boolean values:

$$\text{true, false}$$

together with the primitive operations on these values:

$$\text{not, and, or, if, =, } \neq$$

I.3.5 The List Data Type.

In this section we will define the list data type. The data values are called lists and are written as sequences of values surrounded by angle brackets. For instance,

<5 8 16>

is a list containing the integers five, eight and sixteen, in that order. Lists can have any number of elements, including one:

<32>

This is the list containing only the integer 32. Lists can also be empty, i.e., have no elements:

<>

This is called the null list. Lists can contain any data values, for instance,

<5 'cat' false 1.6>

is a list whose first element is the integer five, whose second element is the string 'cat', whose third element is the Boolean value false, and whose fourth and last element is the real number 1.6. Lists can also contain other lists, for instance,

<5 <9 32> 8 16>

is a list whose first element is the integer five, whose second element is the list <9 32>, whose third and fourth elements are 8 and 16. Lists can be nested in this way to any depth.

We will define three important primitive operations on lists. The function 'first' returns the first element of a list, e.g.,

```
first( <5 8 16> )  =>  5
first( <<1 2> <3 4>> )  =>  <1 2>
```

In the second example notice that the first element of <<1 2> <3 4>> is the list <1 2>. It makes no sense to apply 'first' to the null list or to an atom, which is what we call things that are not lists:

```
first( <> )  is meaningless
first( 18 )  is meaningless
```

Null lists and atoms don't have a first element.

A complementary operation to 'first' is 'rest', which returns all of a list except the first element, e.g.,

```
rest( <5 8 16> )  =>  <8 16>
rest( <<1 2> <3 4>> )  =>  <<3 4>>
rest( <3> )  =>  <>
```

It makes no sense to apply 'rest' to null lists or atoms:

```
rest( <> )  is meaningless
rest( 18 )  is meaningless
```

The 'first' and 'rest' operations take lists apart; we need another operation to put them together. This is 'cons' (short for "construct"), which makes its first argument the new first element of the list which is its second argument. For instance,

```
cons( 5, <8 16> )  =>  <5 8 16>
cons( <1 2>, <<3 4>> )  =>  <<1 2><3 4>>
cons( <5>, <8 16> )  =>  <<5> 8 16>
cons( 5, <> )  =>  <5>
```

Notice that the first argument to 'cons' does not have to be an atom, although its second argument does have to be a list:

```
cons( 5, 8 )  is meaningless
```

The meaning of 'first', 'rest' and 'cons' is summarized in the following formulas. In these, 'x' represents any data value.

```
first( <x ...> )  =>  x
rest( <x ...> )   =>  <...>
cons( x, <...> )  =>  <x ...>
```

These operations can be combined, for instance,

```
first( rest( <5 8 16> ) )
=>  first( <8 16> )
=>  8
```

Hence, first(rest(L)) is the second element of L.

The 'cons' operation is the inverse of 'first' and 'rest'. For example, since

```
first( <5 8 16> )  =>  5,  and
rest( <5 8 16> )  =>  <8 16>,  and
cons( 5, <8 16> )  =>  <5 8 16>
```

we see that

```
cons( first(<5 8 16>), rest(<5 8 16>) )
=>  cons( 5, <8 16> )
=>  <5 8 16>
```

Thus, the following identities hold (where L is a list and x is any value):

```
first(cons(x,L)) = x
rest(cons(x,L)) = L
cons(first(L),rest(L)) = L, if L is non-null
```

Another primitive that will be useful to us is the equality relation. For instance,

```
5=5  =>  true
5=6  =>  false
```

The equality relation is only defined for atoms (e.g., numbers, strings and Boolean values); its use on lists is meaningless:

```
5=<6>  is meaningless
```

The meaning of the equality relation is summarized by the following formulas, in which 'a' and 'b' represent different atoms:

```
a=a  =>  true
a=b  =>  false
```

There are only two other functions that we need to do useful list processing. These are 'null', which asks if a list is empty, and 'atom', which asks whether something is a list. For instance,

```
null( <> )  =>  true
null( <5 8 16> )  =>  false
null( <<>> )  =>  false
```

The list <<>> is not null because it contains a single element, namely, the null list, <>. The 'null' function is not defined for atoms:

```
null(5)  is meaningless
```

Null is defined by the following formulas, in which x is any value:

```
null( <> )  =>  true
null( <x ...> )  =>  false
```

The 'atom' function determines whether a value is an atom or a list, for example,

```
atom( 5 )  =>  true
atom( <5 8 16> )  =>  false
atom( <5> )  =>  false
atom( <> )  =>  false
```

The 'atom' function, which is meaningful when applied to any values, is defined by the following formulas, in which 'a' represents any atom:

```
atom( <...> )  => false
atom( a )  =>  true
```

The list data type is summarized in the following figure.

```
Data Values:
  atoms:  1, 2, ..., 'cat', 'hat', ..., true, false, ....
  lists:  <>, <1 'cat'>, <'call' <'var' 'f'> 23>, ....

Primitive Operations:
  first( <x ...> )  =>  x
  rest( <x ...> )   =>  <...>
  cons( x, <...> )  =>  <x ...>

  a=a   =>  true
  a=a'  =>  false

  null( <> )  =>  true
  null( <x ...> )  =>  false

  atom( a )  =>  true
  atom( <...> )  =>  false

  where a is an atom and x is a list or an atom.
```

Figure 1. The List Data Type

I.3.6 Recursive Problem Solving.

In order to better understand these list processing operations, a number of examples will be presented. First we will define a function 'sub' such that sub(L,i) is the i-th element of the list L. For instance,

```
sub( <5 8 16 25>, 3 )  =>  16
sub( <5 <9 32> 16>, 2 )  =>  <9 32>
```

since <9 32> is the second element of <5 <9 32> 16>. How are we going to program this function? The way to solve problems like this is to ask what subcases of the problem are already solved. In the case of 'sub' this is fairly easy, since by definition

'sub(L,1)' is just the first element of L. That is,

sub(L,1) => first(L)

The next step in solving a problem such as this is to find some way to reduce the general problem to the subcases that are already solved. Notice that

```
sub( <5 8 16 25>, 3 )  =>  16
sub(   <8 16 25>, 2 )  =>  16
```

Hence,

sub(L,i) => sub(rest(L), i-1)

We have reduced the original problem, sub(L,i), to one that is closer to the solved problem, sub(L,1), since i-1 is closer to 1 than i is. We now have two cases, just as in the factorial example:

```
sub(L,i) => first(L),              if i=1
sub(L,i) => sub( rest(L), i-1 ),   if i>1
```

This is easily translated to the lambda calculus:

sub => λLi{ [i=1 -> first(L) | sub(rest(L),i-1)] }

We will try this on sub(<A B C D>, 3):

```
sub(<A B C D>, 3)
=> λLi{ [ i=1 -> first(L) | sub(rest(L),i-1) ] }(<A B C D>, 3)
=> [ 3=1 -> first(<A B C D>) | sub(rest(<A B C D>),3-1) ]
=> sub(<B C D>, 2)
=> [ 2=1 -> first(<B C D>) | sub(rest(<B C D>),2-1) ]
=> sub(<C D>, 1)
=> [ 1=1 -> first(<C D>) | sub(rest(<C D>), 1-1) ]
=> first(<C D>)
=> C
```

The 'sub' function is so useful that we will adopt a special "array subscripting" notation for it:

A[i] => sub(A,i)

Thus, A[1] and first(A) are the same.

Next we will define a function 'append' such that append(L,M) concatenates the lists L and M. That is,

append(<1 2>, <3 4 5>) => <1 2 3 4 5>

Note that this is different from cons(<1 2>,<3 4 5>) which would give us <<1 2> 3 4 5>. We will use the same problem solving technique that we used with 'sub' by asking: What cases of 'append' are immediately solvable? Those in which one of the lists to be appended is null, for instance,

append(<>, <4 5 6>) => <4 5 6>

In general,

append(<>, L) => L
append(L, <>) => L

Next we must investigate how the general problem can be reduced to either of these two cases. For instance, since we know append(<>,L) is L, we can work on reducing the first argument to an empty list. Suppose we wish to simplify

append(<2>, <3 4 5>)

We know

append(<>, <3 4 5>) => <3 4 5>
cons(2, <3 4 5>) => <2 3 4 5>

so we can see that

append(<2>, <3 4 5>)
=> cons(2, <3 4 5>)
=> cons(2, append(<>, <3 4 5>))

Now let's consider a more complicated example:

append(<1 2>, <3 4 5>)

We already know how to do

append(<2>, <3 4 5>) => <2 3 4 5>

so all we have to do is reduce the new case to this:

append(<1 2>, <3 4 5>)
=> cons(1, <2 3 4 5>)
=> cons(1, append(<2>, <3 4 5>))

Summarizing, we have

append(<1 2>, <3 4 5>) => cons(1, append(<2>, <3 4 5>))
append(<2>, <3 4 5>) => cons(2, append(<>, <3 4 5>))

It should be apparent that the general case is

```
append(x,y)  =  cons( x[1], append( rest(x), y ))
```

Summarizing, we have the two cases:

```
append(x,y) => y,                                    if x is null
append(x,y) => cons(x[1], append(rest(x),y)),        if x is non-null
```

This can be easily translated to the lambda calculus:

```
append  =>  λxy{ [ null(x) -> y
                 | cons( x[1], append( rest(x), y)) ] }
```

We will find that all recursive definitions fit this pattern: (1) a stopping condition that forms the base of the recursion and (2) a recursive invocation of the function in which the problem is reduced to a simpler problem. In list processing the stopping condition often takes the form 'null(...)', just as in numerical functions it often takes the form '...=0'. You probably will recognize a similarity between recursive functions and mathematical proofs by induction. This similarity simplifies prooving that recursive functions are correct.

It should be mentioned that if we had decided to reduce the second argument to the null list rather than the first, we would have found the going much tougher. The list processing primitives favor working on the beginning of a list, hence the first argument of 'append'. This sort of intuition comes from practice with using the primitives.

Although it is relatively easy to prove that 'append' is correct, we will convince ourselves by working the reduction of append(<1 2>,<3 4 5>).

```
append(<1 2>,<3 4 5>)
    =>  [ null(<1 2>) -> <3 4 5>
        | cons( <1 2>[1], append( rest(<1 2>), <3 4 5>)) ]
    =>  cons(1, append(<2>, <3 4 5>))
    =>  cons(1, [ null(<2>) -> ...
                | cons(<2>[1],append(rest(<2>),<3 4 5>))]
    =>  cons(1, cons(2, append(<>, <3 4 5>) ))
    =>  cons(1, cons(2, [null(<>) -> <3 4 5> | ... ] ))
    =>  cons(1, cons(2, <3 4 5> ))
    =>  cons(1, <2 3 4 5> )
    =>  <1 2 3 4 5>
```

As final example, we will define the function 'equal' which returns 'true' if its two arguments are equal atoms or lists (i.e. have the same structure). (Recall that '=' only works on

atoms, hence it cannot be used to compare lists.) That is

```
equal( <5 <8 16> 25>, <5 <8 16> 25> )  =>  true
equal( <5 <8 16> 25>, <5 <8> 16 25> )  =>  false
```

As in the previous examples, the stopping point of our recursion will be those cases that we can already solve. Then, we will attempt to reduce the general case to these simpler cases. A good place to begin in any list processing problem is the null list, and this is the case here; all null lists are equal:

```
equal( <>, <> )  =>  true
```

Another good place to begin in list processing is with atoms. This is particularly true in this case, since the '=' relation can be used to test equality of atoms. For example,

```
equal(5,5)  =>  5=5  =>   true
equal(5,6)  =>  5=6  =>  false
```

These cases can be written more generally:

```
equal(x,y) => true,  if x and y are both null
equal(x,y) => false, if x or y is null, but not both

equal(x,y) => true,  if x and y are atoms and x=y
equal(x,y) => false, if x and y are atoms and x≠y
equal(x,y) => false, if x or y is an atom, but not both
```

It remains to reduce the general case to these solved cases. Consider 'equal(x,y)', where x and y are non-null lists (if they aren't then the problem is solved). Now, if x and y are non-null lists, what are the conditions that must be satisfied to make them equal? Clearly, they must have the same number of elements and each of their elements must be 'equal'. Since both lists are non-null we know that they both have a first element. Hence, we can compare the first elements with 'equal', delete them from the lists and then compare the rest of the lists with 'equal'. This is the simplification:

```
equal(x,y) => equal(x[1],y[1]) and equal(rest(x),rest(y)),
              if x and y are non-null lists.
```

Putting our results together allows a direct translation to the lambda calculus:

```
equal  =>  λ xy{
             [ atom(x) ->
                 [ atom(y) -> x=y | false ]
             | atom(y) -> false
             | null(x) -> null(y)
             | null(y) -> false
             | equal(x[1],y[1]) -> equal(rest(x),rest(y))
             | false ] }
```

The 'atom' tests are done first since they work on all values (atoms and lists) while 'null' only works on lists.

EXERCISE: Write out the reduction of equal(<5 <8 16> 25>, <5 <8 16> 25>).
EXERCISE: Define the function 'member' such that member(x,y) is true if and only if x is an element of the list y. That is member(C, <A B C D>) => true, but member(C, <A <B C> D>) => false. To show that your definition works, reduce member(7, <3 5 7 9>).
EXERCISE: Suppose y is a list of pairs, i.e. it has the form

$$\langle\langle a_1\ b_1\rangle\ \langle a_2\ b_2\rangle\ \cdots\ \langle a_n\ b_n\rangle\rangle$$

Define the function 'assoc(x,y)' to be the first b_i for which $x=a_i$. For instance,

```
assoc( 'b', <<'a' 1> <'b' 3> <'c' 5>> )  =>  3
assoc( 7, <<1 0> <7 9> <2 6> <7 3>> )  =>  9
```

EXERCISE: Suppose x and y are lists of the same length and a is a list of pairs. For instance,

$$x = \langle x_1\ x_2\ \cdots\ x_m\rangle$$
$$y = \langle y_1\ y_2\ \cdots\ y_m\rangle$$
$$a = \langle\langle b_1\ c_1\rangle\langle b_2\ c_2\rangle\ \cdots\ \langle b_n\ c_n\rangle\rangle$$

Define 'pairlis(x,y,a)' to be the list resulting from adding the pairs $\langle x_i\ y_i\rangle$ to the front of a, e.g.,

$$\langle\langle x_1\ y_1\rangle\ \cdots\ \langle x_m\ y_m\rangle\ \langle b_1\ c_1\rangle\ \cdots\ \langle b_n\ c_n\rangle\rangle$$

EXERCISE: Write the list processing primitives (first, rest, cons, atom, null) using linked lists in Pascal or some other language you are familiar with.

I.3.7 Syntactic Sugar.

With the list processing primitives we really have a small, but usable, programming language. In fact, the language we have

is very similar to the list processing language LISP, which we will be discussing later. To make the programming language aspects of the lambda calculus more obvious we will "sugar" it with an Algol-style syntax. First, we will replace 'λ' with 'proc' (for 'procedure'), we will write the bound variables in parentheses separated by commas (i.e. 'xy' becomes '(x,y)'), '{' and '}' become 'begin' and 'end', '[' and ']' become 'if' and 'endif', '->' becomes 'then' and '|' becomes either 'elsif' or 'else'. When we make these substitutions in the 'equal' function we get:

```
Equal  =>  proc(x,y)
  begin
    if atom(x) then
      if atom(y) then x=y else false endif
    elsif atom(y) then false
    elsif null(x) then null(y)
    elsif null(y) then false
    elsif equal(x[1], y[1]) then equal( rest(x), rest(y))
    else false
    endif
  end
```

These conventions are summarized in the following diagram.

λxy	=>	proc(x,y)
{...}	=>	begin ... end
[	=>	if
->	=>	then
|	=>	else or elsif
]	=>	endif

We will use the acronym 'ELC' to refer to the extended lambda calculus, i.e. to the pure lambda calculus extended by the integer, Boolean and list data types and extended by the syntactic sugar introduced in this chapter.

ELC is quite similar to the programming language LISP. Approximate equivalences are shown in the following chart (the differences are fairly subtle and are discussed in a later chapter).

ELC	LISP
f(a,b)	(f a b)
λxy{...}	(LAMBDA (x y) ...)
<1 <2 3> 4>	(QUOTE (1 (2 3) 4)
first(x)	(CAR x)
rest(x)	(CDR x)
cons(x,y)	(CONS x y)
[c -> ... | ...]	(COND (c ...) (T ...))
x=y	(EQ x y)
true	T
false	NIL

The LISP program corresponding to our 'equal' function is:

```
(equal (lambda (x y)
  (cond
    ((atom x) (cond
                ((atom y) (eq x y))
                (T NIL)) )
    ((atom y) NIL)
    ((null x) (null y))
    ((null y) NIL)
    ((equal (car x) (car y))  (equal (cdr x) (cdr y)) )
    (T NIL) )))
```

You can now see that languages can have very different appearances even though they are the same underneath. We will find that most programming languages are sugared versions of the lambda calculus.

I.3.8 local declarations in mathematical prose. If a mathematician were to write an expression such as

$$\frac{\left(\frac{ax+b}{x}-a\right)^{m+n-2}}{\left(\frac{ax+b}{x}\right)^{m}}$$

he would probably factor out the common subexpression

$$\frac{ax+b}{x}$$

and give it a name, say 'u':

$$\frac{(u-a)^{m+n-2}}{u^{m}} \qquad \text{where} \qquad u = \frac{ax+b}{x}$$

This has two advantages. First, it reduces the size and complexity of each expression so that they can be more easily assimilated by the eye. Second, by using the same variable 'u' in both places it makes it more obvious that it is the same identical

expression in both places. Now, we can accomplish the same thing in the lambda-calculus. In particular,

$$\lambda u\left\{\frac{(u-a)^{m+n-2}}{u^m}\right\}\left(\frac{ax+b}{x}\right)$$

reduces to

$$\frac{\left(\frac{ax+b}{x}-a\right)^{m+n-2}}{\left(\frac{ax+b}{x}\right)^{m}}$$

I.3.9 local declarations in the lambda calculus The "where" notation will be defined as a "syntactic sugaring" of function application:

$$E \text{ where } v=x \;\Rightarrow\; \lambda v\{E\}(x)$$

For instance, an expression such as this (which is taken from the interpreter discussed later):

```
eval( f[1][2], append( f[2], cons(x,<>) ))
```

could now be written

```
eval( f[1][2], NewEnv )
  where NewEnv = append( f[2], cons(x,<>))
```

By analogy with functions of several arguments, it is possible to allow compound "where" declarations:

$$E \text{ where } v_1=x_1 \text{ and } v_2=x_2 \text{ and } \cdots \;\Rightarrow\; \lambda v_1 v_2 \cdots \{E\}(x_1,x_2,\ldots)$$

For instance, the expression fragment

```
elsif e[1] = 'var' then a[e[2]][e[3]]
```

can be more readably written

```
elsif e[1] = 'var' then a[ep][vp]
    where ep = e[2]
    and   vp = e[3]
```

There is another way in which mathematicians use variable names. If a mathematician is going to use an expression in a number of following lines then he will often give it a name with a phrase such as "Let u be $\frac{ax+b}{x}$." This style of definition is easily defined in the lambda-calculus:

```
let v=x in E  =>  λv{E}(x)
```

or in general

```
let v1=x1 and v2=x2 and ··· in E
=>  λv1v2···{E}(x1,x2,...)
```

Whether 'let' or 'where' is used is largely a matter of taste and style. In general, 'where' is appropriate when 'E' is one line or less and 'let' is appropriate when 'E' is more than one line. As an example of 'let', the expression fragment

```
elsif e[1] = 'call' then
  apply( eval( e[2], a), evlis( rest(rest(e)), a) )
```

can be written

```
elsif e[1] = 'call' then
  let closure = eval(e[2],a)
  and actuals = eval( rest(rest(e)), a) in
    apply( closure, actuals )
```

The use of names such as 'closure' and 'actuals' makes the program more readable, albeit, more verbose.

I.3.10 terminology It will be recalled that in the abstraction 'λx{E}' the scope of 'x' was defined to be 'E'. By analogy, the scope of the variables defined by a 'let' or a 'where' is defined to be the body of the corresponding abstraction. For instance, in the previous example, the scope of 'closure' and 'actuals' is the following line. To put it another way, the scope of a variable is the region of an expression over which it has meaning. This is shown in the following diagrams:

```
λv{E}
     ←— scope of 'v'

E where v=x
  ←———— scope of 'v'

let v=x in E
          scope of 'v'
```

Consider the following 'let' or 'where' expressions:

```
E where v1=x1 and v2=x2 and ···
let v1=x1 and v2=x2 and ··· in E
```

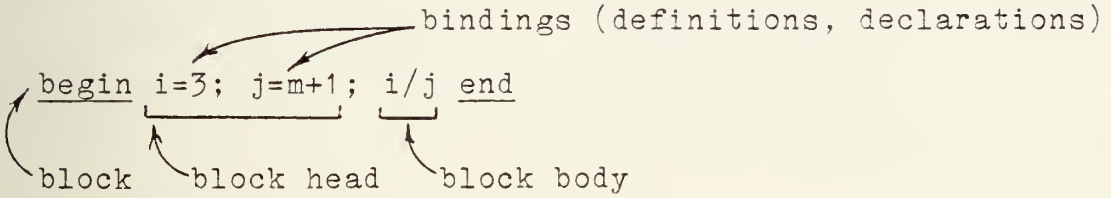
The '$v_i=x_i$' parts of these are called *definitions*, *declarations* or *bindings* (because they bind a bound variable to its value). The 'let' and 'where' constructs themselves are called *blocks* (particularly in languages that delimit them with *begin*-*end* pairs). These constructs occur in most languages although in a variety of forms. For instance, the Algol-68 block

```
begin i=3; j=m+1;
      i/j
end
```

means the same as the lambda-calculus block

```
let i=3 and j=m+1
in i/j
```

In languages that use *begin*-*end* pairs to delimit blocks, like Algol-68, the declarations ('i=3; j=m+1' in this case) are collectively known as the *head* of the block, and the rest of it is known as the *body* of the block. Thus,



A set of Pascal declarations such as

```
function g(y:integer): integer;
  const i=3;
  function f(x:integer): integer;
    begin  f := x*i+1  end;
  begin  g := f(y)+1  end;
begin ... end
```

means the same as the lambda expression

```
let g = proc(y) begin
  let i=3 in
    let f = proc(x) begin x*i+1 end in
      f(y)+1  end
in ...
```

Of course, it is necessary to specify types in Pascal declarations, but not in lambda-calculus bindings.

As a final example of blocks in the lambda-calculus, consider the lambda expression

$$\lambda i\{\ \lambda f\{\ \lambda i\{f(i)\}(2)\ \}(\ \lambda x\{x*i\}\ )\ \}(3)$$

The meaning of this expression can be clarified by using blocks:

```
let i = 3 in
  let f = proc(x){ x*i } in
    let i = 2 in
      f(i)
```

Although there is no exactly corresponding Pascal program, this lambda expression can be translated into Algol-68:

```
begin i = 3;
  begin f = proc(x:int)int: (x*i);
    begin i = 2;
      f(i)
    end
  end
end
```

I.3.11 compound declarations Consider the two expressions

let v_1=e_1 in let v_2=e_2 in e_3

scope of v_2

scope of v_1

let v_1=e_1 and v_2=e_2 in e_3

scope of both v_1 and v_2

The second of these, which is called a compound declaration, is much more than merely a shorthand form of the first. Observe that in the first expression e_2 is within the scope of v_1, whereas in the second expression the scope of both v_1 and v_2 is just e_3. For instance, while

```
let i=3 in let x=A[i] in B
```

has the same effect as

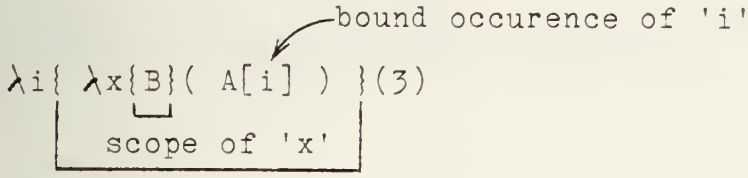
```
let x=a[3] in B
```

the expression

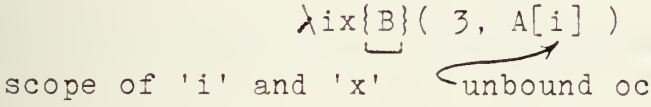
```
let i=3 and x=a[i] in B
```

is illegal (i.e. doesn't make any sense), unless, of course, 'i'

is bound in some surrounding scope. This can be seen more clearly by translating the two expressions back to pure lambda notation. The first becomes:



which is correct, and the second becomes:



which is not. The most important use of compound declarations is in connection with recursive declarations, discussed next.

I.3.12 recursive declarations Consider a declaration such as

```
let fac = proc(n){ [n=0 -> 1 ¦ n*fac(n-1)] }
in B
```

scope of 'fac'

This is illegal, as we can see by putting it in pure lambda notation:

```
λfac{B}( λn{ [n=0 -> 1 ¦ n*fac(n-1)] } )
```

scope of 'fac' — unbound occurence of 'fac'

The recursive occurence of 'fac' is unbound. This is because the value on the right of a binding is evaluated in the surrounding environment. This is clearly an unsatisfactory situation. Although there are ways of handling recursive definitions in a purely applicative way, they involve complicated mathematics and so will not be discussed here. Instead, the technique to be used will require some of the imperative facilities discussed in the next chapter. Although the actual technique to be used will not be described until the next chapter, recursive definitions will be used in this chapter.

A declaration of the form

```
let rec v=e in B
        └──────┘
    scope of 'v'
```

is called a recursive declaration (or definition or binding) because the scope of 'v' includes 'e'. Therefore 'e' can make use of 'v'. Similarly, compound recursive declarations will be defined so that in

```
let rec v1=e1
and rec v2=e2
and ... in B
```

scope of v_1, v_2, ...

the scopes of v_1, v_2, ... include e_1, e_2, ... This allows the definition of mutually recursive functions, for instance:

```
let rec Eval = proc(e,a){ ... apply(...) ... }
and rec Apply = proc(f,x){ ... eval(...) ... }
in ...
```

CHAPTER 4

I.4 Implementing The Lambda Calculus

I.4.1 goals defined.

You have probably found reducing lambda calculus expressions to be a tedious and error-prone process. These characteristics, plus the fact that reduction is a mechanical procedure, makes the computer the ideal tool for doing these reductions. There are several reasons for this. When people work for a long time at a mechanical task, they become tired and begin to make mistakes. Computers aren't like that: once programmed correctly they will perform a task without fatigue. Another advantage of using computers to do reductions is that they can do them so fast. The mechanical, symbolic processes of a calculus are exactly what computers handle best, since a computer is just a very fast symbol manipulator.

Why are we so concerned about reducing lambda expressions, anyway? As you've seen in the previous chapter, languages with very different appearances are often just "sugared" versions of the lambda calculus. You've also been told (and you will see it in Part II) that most common programming languages are, underneath, just the lambda calculus. Therefore, if you study how to implement the lambda calculus, you will be learning how to implement most of the common programming languages. The advantage of using the lambda calculus, as opposed to Pascal or Ada, for instance, is that the lambda calculus is so simple. The implementation techniques are much easier to understand in the clear and uncluttered context of the lambda calculus. Once understood, they are easy to extend or modify so that they accomodate the complexities of "real" languages.

I.4.2 mechanical reduction

Having established that imolemenation of the lambda calculus is important, we can proceed to study how we might go about it. The obvious approach is to write a program that duplicates our hand reduction procedures. This is easier said than done, however.

Let's investigate how we might go about automating the reduction process. When we reduce a lambda expression using the Standard Reduction Order, we do it by applying the substitution rule over and over, until we are forced to stop by an immanent

collision of variables. We then use the renaming rule to change the offending bound variable to one that won't cause a collision, and continue with our substitutions. Except on very complex lambda expressions it is generally easy to see if there will be a collision of variables. This sort of pattern recognition process is the sort of thing that is very easy for people and very hard for computers. This is because we can see the collision at a glance, while the computer must do the check with an exhaustive search. That is, in order to determine if it is legal to substitute an actual parameter for a particular (free) occurence of a variable, it is first necessary to compute a list of the free variables in the actual parameter. Doing this requires scanning the expression and noting all the binding sites and their scopes. It is then necessary to compute all the variables whose scopes contain the prospective substitution site, and to determine if any of these variables are the same as free variables of the actual parameter. This is a expensive check to perform and intricate and tricky to program.

Another reason to avoid a mechanical implementation of the reduction procedure is that it would be slow. As you've probably observed in your own reductions, this process involves a lot of recopying of the formulas. This is something that most computers don't do well. For these reasons we will investigate an implementation that involves neither textual substitution nor a complicated collision-of-variables test.

I.4.3 context

Let's take another look at the way people use variables. Suppose a mathematician reads a phrase such as, "Let $\theta = 2\pi ft$." What happens when he later reads a formula such as "$2\sin\theta\cos\theta$"? Does he mentally transform this into

$$2\sin(2\pi ft)\cos(2\pi ft)$$

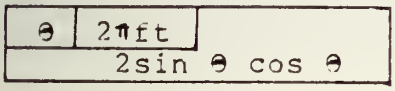
by performming the substitutions? Of course not. Having been told "Let $\theta = 2\pi ft$", he remembers this fact and associates θ with $2\pi ft$. We say that he has bound θ to $2\pi ft$. When he reads the formula "$2\sin\theta\cos\theta$" he interprets it contextually, i.e., in the context of the relevent meanings of 2, sin, cos and θ. It is not necessary for him to do a textual substitution.

You will remember that the reason for the collision-of-variables restriction on the substitution rule was to prevent altering the meaning of an expression by changing the context of its interpretation. In this case the context is just a set of bindings, i.e., associations between variables and their values. In the next section you will see how to keep track of the context of a formula in such a way that it can be evaluated without the

use of substitution, renaming or collision tests.

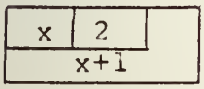
I.4.4 hand evaluation

In this section we will study the use of the context or environment of a formula - i.e., the set of bindings that give that formula its meaning. These will be represented by diagrams of the form:

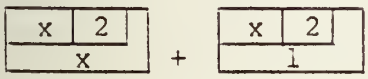


This expreses the fact that the context of the formula '2sin θ cos θ' is the binding of θ to 2πft.

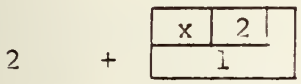
As our first example, let's consider the evaluation of 'x+1' in a context in which x is bound to 2, i.e.,



Since this is the context of the entire formula, it is also the context in which each of its subformulas must be interpreted. Hence, this can be reduced to



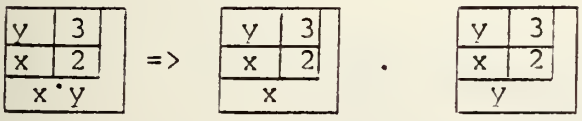
Now, the interpretation of x in the context x=2 is just 2, so we have



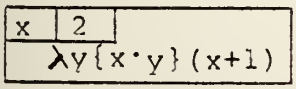
and the interpretation of 1 in any context is 1, so

2 + 1

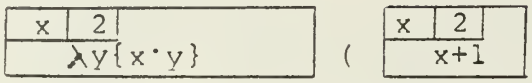
or 3. Similarly, we can evaluate x·y in the context y=3, x=2.



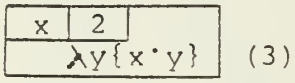
Next, we'll take a more complicated example, the evaluation of 'λy{x·y}(x+1)' in the environment x=2:



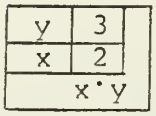
Before we can bind y to x+1 we have to know the value of x+1 (in context, of course). Hence, we will separate the two parts of the application:



Notice that we have kept the formula λy{x·y} in context - otherwise it would lose its meaning. Continuing our evaluation, we previously saw that x+1 in the context x=2 evaluates to 3, so we have:

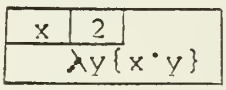


What is the effect of λy{x·y}(3)? It is to add the binding y=3 to the context of interpretation:

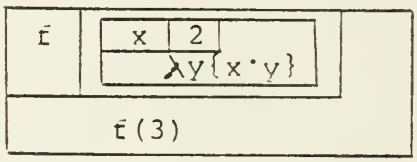


We have previously seen that this reduces to 6.

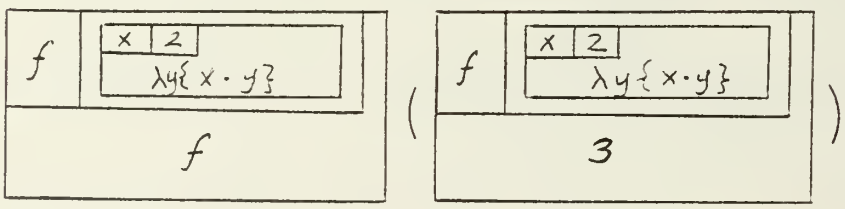
We will expand on the previous example a little. In that example we applied to the argument 3 the function λy{x·y} in the context x=2, i.e., the function represented by



Next we will consider the evaluation of f(3) in a context that binds f to the above function. The result, of course, should be the same. We start with

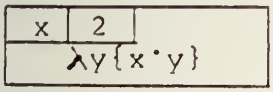


We must evaluate the operator and operand separately:



Constants are independent of the context, so the operand's value

is 3. The operator is a variable, f, which must be interpreted in the context. When when we substitute this definition we get:

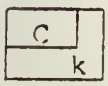


(3)

which we have already seen is 6. Notice that we have always preserved the meaning of λy{x·y} by bringing the context along with it. We are now at the point where we can formally state our evaluation rules.

I.4.5 constants

Since the value of a constant does not depend on its context, we evaluate it by eliminating the context.

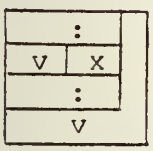


=> k

for k a constant.

I.4.6 variables

In contrast to constants, the value of a variable depends entirely on the context. We find the value of a variable by looking it up in the list of bindings in the contour (by convention, we take the first occurence of the variable in that list). The rule is:



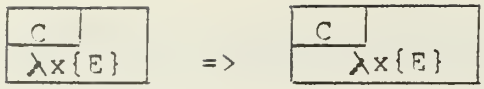
=> x

where v is a variable.

If x is not bound in this context, then it is free (i.e., so far as we know it has no value).

I.4.7 abstractions

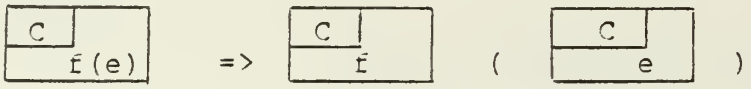
An abstraction often will have free variables that are defined in its context. If this context is not preserved then the abstraction will change its meaning. Therefore, we will leave an abstraction unevaluated until it is ready to be applied to some arguments. We show this delay of evaluation:



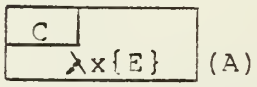
(no change)

I.4.8 applications

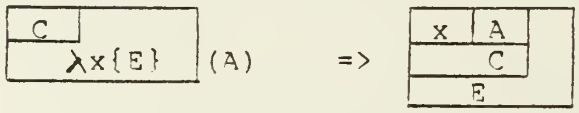
Consider an application such as 'f(x+1)'. Standard Reduction Order says that in order to determine its value it is first necessary to determine the value of its operand, 'x+1', and its operator, 'f'. Of course the value of each of these is found by interpreting it in the context of the entire application:



Of course, there is more to the evaluation of applications than this. If the application is to make sense then the evaluation of the operator must result in an abstraction (with its context). That is, we will get a result like:

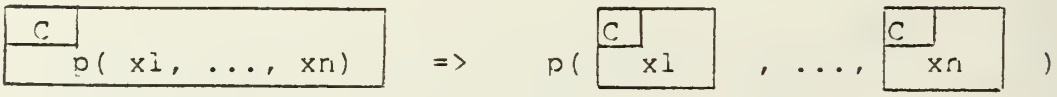


Now, this means that C is the context of the abstraction λx{E}. The application above is evaluated by adding the binding x=A to the context C, and then using this as the context in which to interpret E. Summarizing, the rule is:



I.4.9 primitive applications

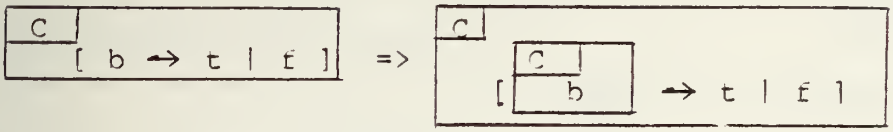
We have already seen several examples of the evaluation of primitive applications, such as 'x+1' and 'x*y'. In each of these it was first necessary to evaluate the operands of the primitive. This leads to the general rule:



I.4.10 conditionals

Recall that we specified a deviation from Standard Reduction Order for conditionals. We said that we would first reduce the condition and then, depending on whether the result was 'true' or 'false', reduce either the true branch or the false branch. This avoids errors in situations where the other branch is not

reducible. Reducing the condition is accomplished by the rule:



When the condition has evaluated to 'true' or 'false' the following rules interpret in context the corresponding branch.

I.4.11 example of hand evaluation

As an example of these procedures we will evaluate

```
let i=3 in
  let f = proc(x){ x•i } in
    let i=2 in
      f(i)
```

which we saw in Chapter 3. The crucial aspect of this example is that the context of proc(x){x•i} is i=3. We can see that this context is preserved in the following figure. In this example we first eliminate syntactic sugar and write the above formula:

$$\lambda i\{\ \lambda f\{\ \lambda i\{\ f(i)\ \}(2)\ \}(\ \lambda x\{x \bullet i\}\ )\ \}(3)$$

I.4.12 representation defined

We will next write an interpreter for ELC (the extended lambda calculus) in ELC. Since we will be using ELC, and the only primitives we have in ELC are for manipulating lists, we will have to encode lambda expressions as lists so that they can be manipulated. This is the representation we will use:

FORM	EXAMPLE	REPRESENTATION
constants	2	<'const' 2>
	'string'	<'const' 'string'>
	<5 8 16>	<'const' <5 8 16>>
variables	x	<'var' 'x'>
abstractions	λxy{E}	<'lambda' <'x' 'y'> E>
applications	F(A,B)	<'call' F <A B>>
prim. applics.	cons(A,B)	<'prim' 'cons' <A B>>
conditionals	[c → t | f]	<'if' c t f>

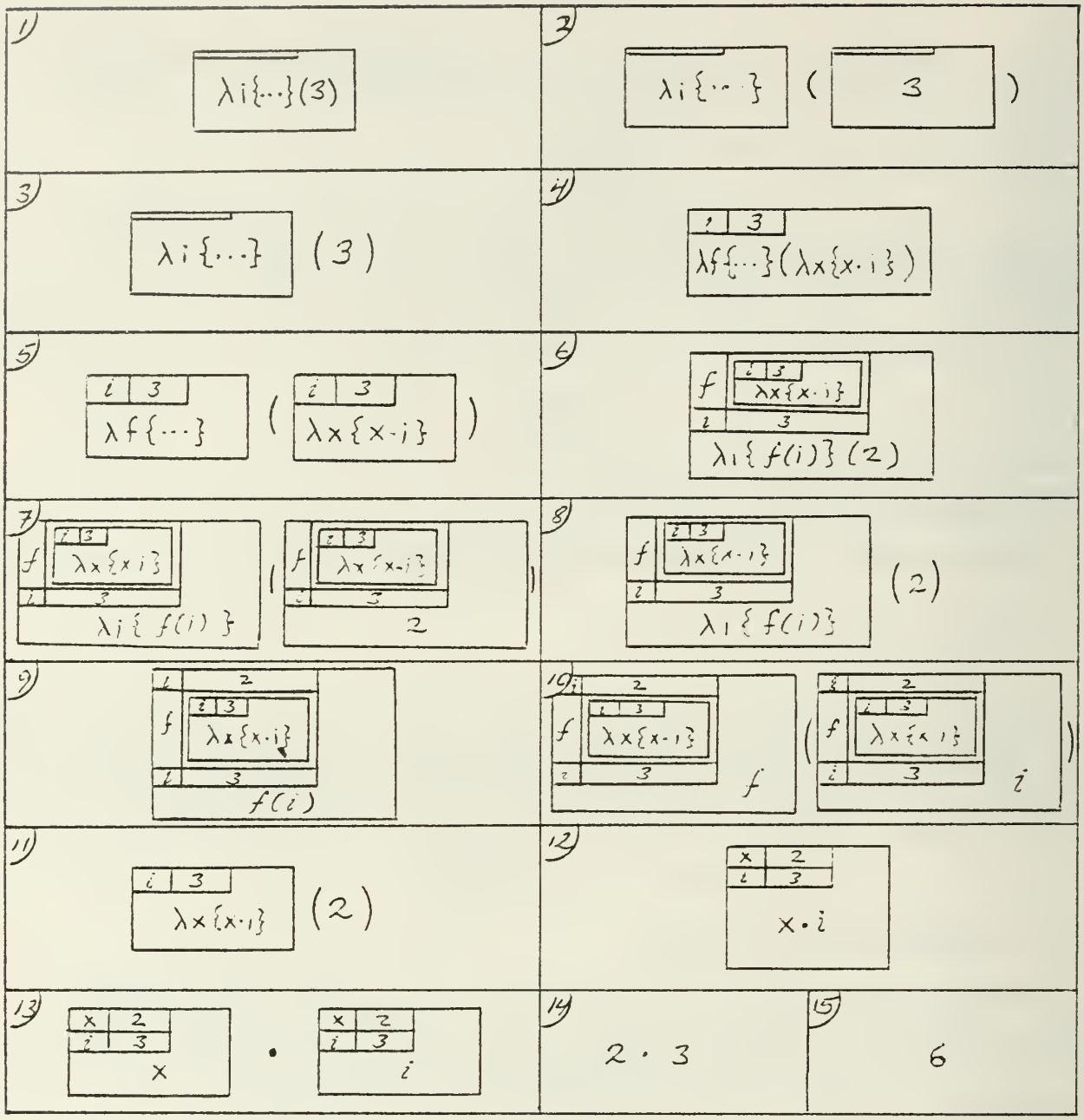


Figure 1. Example of Hand Evaluation

It is necessary to append the "tag word" 'const' to the beginning of constants to prevent confusion, for instance, between the lists representing variables and constant lists that happen to begin with the string 'var'.

We will look at several examples of lists that represent lambda expressions. The expression

f(1,2)

is represented by the list:

<'call' <'var' 'f'> ≪'const' 1> <'const' 2>>>

The primitive application

cons(y,<>)

is represented by

<'prim' 'cons' ≪'var' 'y'> <'const' <>> >>

For a more complicated example, consider:

λxy{ cons(x, cons(y, <>)) }

This is represented by

```
<'lambda' <'x' 'y'>
  <'prim' 'cons' ≪'var' 'x'>
                 <'prim' 'cons' ≪'var' y'>
                                <'const' <>> >>> >>
```

I.4.13 field accessing functions

Suppose L is a list representing an abstraction, e.g.,

<'lambda' <'x' 'y'> <'var' 'x'>>

If we wish to extract the bound variables from L, we can do this by L[2]:

L[2] => <'x' 'y'>

Similarly, the body of the abstraction can be extracted by L[3]:

L[3] => <'var' 'x'>

Both of these operations will be more readable if we define the field accessing functions "bvs" and "body" to get the bound variables and body, respectively, of an abstraction:

```
let bvs = proc(x){ x[2] }
and body = proc(x){ x[3] }  in ...
```

Then we can write

```
bvs(L)  =>  <'x' 'y'>
body(L)  =>  <'var' 'x'>
```

It will also be useful to have a function 'is-lambda' to tell us if a list represents an abstraction:

```
let is-lambda = proc(x){ x[1] = 'lambda' } in ...
```

This function determines if a list represents an abstraction by checking if its "tag word" is 'lambda'. For instance,

```
is-lambda(L)  =>  true
is-lambda( <'const' 2> )  =>  false
```

Of course, we will want functions like 'body' and 'is-lambda' for each of the lists which represent a type of lambda expression. Rather than define each separately like we did above, we will use the abbreviation

structure lambda: (bvs, body)

to mean that lists whose first element is 'lambda' will have two more elements, selected by the functions 'bvs' and 'body'. This is called a structure definition and is equivalent to the declaration:

```
let is-lambda = proc(x){ x[1]='lambda' }
and bvs = proc(x){ x[2] }
and body = proc(x){ x[3] }
in  ...
```

The structure declarations for the lists that represent lambda expressions are given in the following figure. The definition of these field accessing functions will make our interpreter much more readable.

```
structure const: (constval)
structure var: (id)
structure lambda: (bvs, body)
structure call: (rator, rands)
structure prim: (rator, rands)
structure if: (cond, t-branch, f-branch)
```

Figure 2. Lists Representing Lambda Expressions

I.4.14 association lists.

So that environments (contexts) can be manipulated by the primitives with which we have equipped the lambda calculus, we will define a representation, called an association list (or a-

list), for environments. An association list is just a list wherein each element of the list is a pair containing the name of a variable and its value. For instance, an association list representing an environment in which x=2, y='Monterey' and z=<0 1 2> is:

```
< <'x' 2> <'y' 'Monterey'> <'z' <0 1 2>> >
```

We will define several primitive functions on association lists. The function 'assoc' is defined so that 'assoc(x,a)' is the result of looking up that variable name x in the association list a.

```
let rec assoc = proc(x,a) begin
  if equal( x, a[1][1]) then a[1][2]
  else assoc( x, rest(a)) endif end.
```

For instance, if L is the a-list

```
< <'x' 2> <'y' 'Monterey'> <'z' <0 1 2>> >
```

then

```
assoc( 'y', L) => 'Monterey'.
```

The function 'pairlis' is a little more complicated; it is used for binding variables to values and adding them to an association list. That is, if x is a list of variable names, y is a list of values and a is an association list, then 'pairlis(x,y,a)' is an association list in which each element of x is paired with the corresponding element of y and appended to the front of a. For instance,

```
pairlis( <'w' 'cost'>, <6 25>, L )
=>  <<'w' 6> <'cost' 25> <'x' 2> <'y' 'Monterey'> <'z' <0 1 2>>>.
```

To see how this can be done, notice that if 'a' is an a-list, and <xi yi> is a pair representing a binding, then

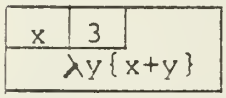
```
cons( <xi yi>, a)
```

will add this pair to 'a'. The resulting definition of 'pairlis' is:

```
let rec pairlis = proc(x,y,a) begin
  if null(x) then a
  else cons( pair( x[1], y[1] ),
             pairlis( rest(x), rest(y), a) )  endif end
  where pair = proc(x,y){ cons( x, cons( y, <> )) }.
```

I.4.15 closures

We have previously seen that it is necessary to keep an abstraction's context with that abstraction in order to preserve its meaning. We have symbolized this with diagrams like:



We will now investigate in more detail the nature of this construct.

A formula is called closed if it has no free variables, for instance,

$$\lambda x\{\ \lambda y\{x+y\}\ \}(3)$$

Closed formulas are important because their interpretation is completely independent of context. A formula is called open if it has one or more free variables. Hence, open formulas are dependent on their context for their interpretation. This formula is open:

$$\lambda y\{x+y\}$$

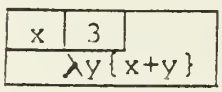
An open formula can be closed by providing bindings for its free variables. For instance, the above open formula can be closed by binding x to 3:

$$\underline{\text{let}}\ x=3\ \underline{\text{in}}\ \lambda y\{x+y\}$$

or

$$\lambda x\{\ \lambda y\{x+y\}\ \}(3)$$

This is called the closure of the formula $\lambda y\{x+y\}$. More generally, a closure is a formula together with bindings for all of its free variables. This is just the construct we have illustrated by



Our next task will be to determine how we can implement closures using lists. The key is our diagram; a closure has two parts: an abstraction and its context. These parts are traditionally called the ip and ep, which stand for instruction part and environment part, respectively. We will represent a closure by a structure with two fields called 'ip' and 'ep'. The ip will be a list that represents an abstraction and the ep will be an a-list

representing a context. This is summarized by

structure closure: (ip, ep)

The function for constructing closures is defined:

```
let closure = proc(ip,ep){
  cons( 'closure', pair( ip, ep )) }
```

Thus, we can make a closure from the abstraction f and the environment e by closure(f,e):

closure(f,e) => <'closure' f e>

We can test if something is a closure by is-closure(c) and extract its parts by ip(c) and ep(c). The closure data type is described in the following figure.

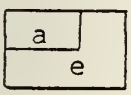
```
ip( closure(f,e) )  =  f
ep( closure(f,e) )  =  e
closure( ip(c), ep(c) )  =  c
is-closure( closure(f,e) )  =  true

where is-lambda(f)
and   e is an a-list
and   is-closure(c)
```

Figure 3. The Closure Data Type

I.4.16 mechanical evaluation.

We will define a function 'eval' such that if 'e' is a list representing a lambda expression and 'a' is an association list representing an environment, then 'eval(e,a)' is the result of evaluating that lambda expression in that environment. Thus, 'eval(e,a)' corresponds to



The structure of our interpreter will be a large conditional so that we can handle lambda expressions case by case. In skeleton form it is:

```
let rec Eval = proc(e,a)
  begin
    if    is-const(e)  then ...
    elsif is-var(e)    then ...
    elsif is-lambda(e) then ...
    elsif is-call(e)   then ...
    elsif is-prim(e)   then ...
    elsif is-if(e)     then ...
    endif
  end
in ...
```

We will consider each of these cases in turn. In discussing them, it will be helpful if you refer back to the field names defined in figure 2.

The simplest case, as we have already seen, is constants, because they do not depend on the context for their interpretation. In this case we just extract the constant value from the list and return it:

```
if is-const(e) then const-val(e)
```

The next simpler case is variables. The result of evaluating <'var' x> is just the result of looking up x in the current environment (which is the a-list bound to 'a'):

```
elsif is-var(e) then assoc( id(e), a)
```

As discussed in the previous sections, the rule for abstractions will form a closure by combining the abstraction and its environment of definition (context). This is accomplished by:

```
elsif is-lambda(e) then closure(e,a)
```

When this closure is applied to its operands by a application, the 'apply' function will extract the environment of definition (ep) from the closure and use it as the basis for constructing the environment of evaluation for the body of the abstraction (ip).

Next we will consider the case of primitive applications. If e is a primitive application then rator(e) is its operator, which is a string such as 'first' or 'cons', and rands(e) is a list of its operands. In our hand evaluation procedure, we first interpreted the operands in context and then performed the primitive operation. Here we will use an auxilliary function 'evlis' to evaluate the operands and an auxilliary function 'apply-prim' to perform the operation:

```
elsif is-prim(e) then apply-prim( rator(e), actuals)
  where actuals = evlis( rands(e), a)
```

Evlis is a simple recursive procedure that evaluates each element of its argument list and returns a list of the results:

```
let rec evlis = proc(L,a) begin
  if null(L) then <>
  else cons( eval( L[1], a), evlis( rest(L), a))  endif end in ...
```

The definition of 'apply-prim' is simply a conditional for handling each of the primitive operations:

```
let apply-prim = proc(f,x) begin
  if    f='first' then first(x[1])
  elsif f='rest'  then rest(x[1])
  elsif f='cons'  then cons( x[1], x[2])
  elsif f='atom'  then atom(x[1])
  elsif f='null'  then null(x[1])
  endif end in ...
```

Of course, if we want additional primitive operations, we can just add clauses for handling them to the definition of 'apply-prim'.

The application of non-primitve operations is a similar process. The operands must be evaluated using 'evlis', as was done for primitive applications. Since the operator is also a formula, it also must be evaluated in context. This evaluation must yield a closure. The closure is then applied to the actual parameters. That is:

```
elsif is-call(e) then apply( closure, actuals)
  where closure = eval( rator(e), a)
  and   actuals = evlis( rands(e), a)
```

Next, we must consider what 'apply' has to do to complete the function call. In our hand evaluation procedure we paired the bound variables with the corresponding actual parameters and added these bindings to the context of the abstraction. This new context was used as the environment for evaluating the body of the abstraction. We will do the same here.

If b is the list of bound variables, x is the list of actual parameters and c is the context of the abstraction, then pairlis(b,x,c) is the environment in which to evaluate the body of the abstraction. The 'apply' function that accomplishes this is:

```
let rec apply = proc( closure, actuals) begin
  let abs = ip(closure) in
    let env = pairlis( bvs(abs), actuals, ep(closure) ) in
      eval( body(abs), env ) end
```

The only case left to be analyzed is the conditional. In our hand evaluation procedure we interpreted the condition in context to get a truth value. We then used this value to determine whether to evaluate the true branch or the false branch of the conditional. This is exactly what needs to be done in 'eval':

```
elsif is-if(e) then
  if eval( cond(e), a)
    then eval( t-branch(e), a)
    else eval( f-branch(e), a)  endif
```

This completes the definition of 'eval'; the complete interpreter is shown in the following figure.

```
let rec Eval = proc(e,a) begin
  if    is-const(e)  then const-val(e)
  elsif is-var(e)    then assoc( id(e), a)
  elsif is-lambda(e) then closure( e, a)
  elsif is-prim(e)   then apply-prim( rator(e), actuals)
    where actuals = evlis( rands(e), a)
  elsif is-call(e)   then apply( closure, actuals)
    where closure = eval( rator(e), a)
    and   actuals = evlis( rands(e), a)
  elsif is-if(e)     then
    if eval( cond(e), a) then eval( t-branch(e), a)
    else eval( f-branch(e), a) endif
  endif end

and rec evlis  =  proc(L,a) begin
  if null(L) then <>
  else cons( eval(L[1],a), evlis(rest(L),a) )  endif  end

and rec apply = proc( closure, actuals) begin
  let abs = ip(closure) in
    let env = pairlis( bvs(abs), actuals, ep(closure)) in
      eval( body(abs), env)  end

and apply-prim = proc(f,x) begin
  if    f='first' then first(x[1])
  elsif f='cons'  then cons( x[1], x[2] )
  elsif ...
  endif end
in ...
```

Figure 4. A-List Eval

The interpreter is very similar to the interpreters used for LISP and similar languages. To clarify its operation, several examples will be presented. In these examples we will trace the invocations of 'eval' and 'apply' and occasionally other functions.

EXAMPLE 1: reduction of λxy{cons(y,x)}(<2 3>, 1).

In the list representation this is:

```
E = <'call' F A>
F = <'lambda' <'x' 'y'> B>
B = <'prim' 'cons' P>
P = <<'var' 'y'> <'var' 'x'>>
A = <<'const' <2 3>> <'const' 1>>

Eval(E, <>):
 closure = eval(F,<>) = <'closure' F <>>
 actuals = evlis(A,<>) = <<2 3> 1>
 apply( closure, actuals ):
  abs = F
  env = pairlis( <'x' 'y'>, actuals, <> )
      = <<'x' <2 3>> <'y' 1>>
  eval(B,env):
   actuals = evlis( <<'var' 'y'> <'var' 'x'>>, env )
           = <1 <2 3>>
   apply-prim( 'cons', <1 <2 3>> ):
    cons( 1, <2 3> ) = <1 2 3>
```

EXAMPLE 2: Reduce

```
let i = 3 in
  let f = λx{ prod(x,i) } in
    let i = 2 in
      f(i)
```

To translate this to list representation it is necessary to first eliminate syntactic sugar:

λi{ λf{ λi{ f(i) }(2)}(λx{prod(x,i)})}(3).

Notice that by doing a hand reduction of this we can determine that the correct reduction is 6. For the sake of this example we will assume that 'eval' can evaluate the primitive application 'prod', which multiplies two numbers.

```
E = <'call' I <<'const' 3>> >
I = <'lambda' <'1'> <'call' F <X>> >
F = <'lambda' <'2'> <'call' J <<'const' 2>> >>
J = <'lambda' <'1'> <'call' <'var' '2'> <<'var' '1'>> >>
X = <'lambda' <'x'> P>
P = <'prim' 'prod' <<'var' 'x'> <'var' '1'>> >

Eval( E, <> ):
 closure = <'closure' I <>>
 actuals = <3>
 apply( closure, actuals ):
  abs = I
  env = <<'1' 3>>
  eval( <'call' F <X>>, <<'1' 3>> ):
   closure = <'closure' F <<'1' 3>> >
   actuals = evlis( <X>, <<'1' 3>> )
           = <C>  where  C = <'closure' X <<'1' 3>> >
   apply( closure, actuals ):
    abs = X
    env = <<'2' C> <'1' 3>>
    eval( <'call' J <<'const' 2>>>, env ):
     closure = <'closure' J <<'2' C> <'1' 3>> >
     actuals = <2>
     apply( closure, actuals ):
      abs = J
      env = <<'1' 2> <'2' C> <'1' 3>>
      eval( <'call' <'var' '2'> <<'var' '1'>> >, env ):
       closure = eval( <'var' '2'>, env ) = C
       actuals = <2>
       apply( closure, actuals):
        abs = X
        env = pairlis( <'x'>, <2>, ep(closure) )
            = pairlis( <'x'>, <2>, <<'1' 3>> )
            = <<'x' 2> <'1' 3>>
        eval( P, env ):
         actuals = eval( <<'var' 'x'> <'var' '1'>>, env )
                 = <2 3>
         prim-apply( 'prod', <2 3> ):
          prod(2,3) = 6
```

Do the reduction by hand to be sure you see that this is the right answer.

I.4.17 replacement of variables by fixed locations. Observe that our current interpreter spends a lot of its time searching association lists for the values of variables. That is, the evaluation of <'var' 'x'> in the environment 'a' requires searching 'a' for a pair whose first element is 'x'. Such searching is expensive on most computers, so we will develop a method of interpreting the lambda calculus which does not require it. In particular, we will replace searching by array subscripting,

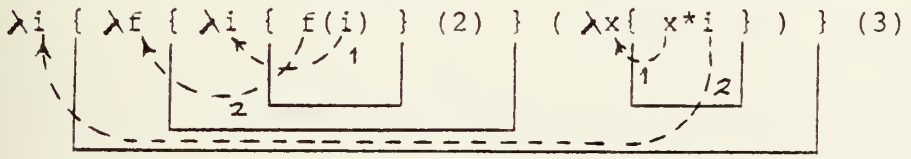
which is much more efficient. We will develop the process for single variable abstractions initially and later extend it for multiple variables. Consider an abstraction 'λx{E}' in some environment '<...>'. The closure formed from evaluating this abstraction will be

<'closure' <'lambda' <'x'> E> <...> >.

When this closure is applied to an operand value 'u' its evaluation will always take the form

eval(E, <<'x' u> ...>).

Notice that the bound variable of the abstraction being evaluated will always occupy the first pair of the environment. Therefore there is really no reason to search for this variable since we know where it is. Since the abstraction we chose was arbitrary, this can be seen to be true for all abstractions. In fact, we find that it is true for all variables that we can determine the position of each variable in the environment by counting the number of "scoping lines" that must be crossed in getting from a use of the variable to its definition. This number is called the static distance of a variable from its definition, and is formally defined to be the number of scopes containing the use of the variable that do not contain its definition (binding occurence). For instance, in



the static distances of the variables from their definitions are indicated. For 'f' it is 2, for 'x' it is 1, for the leftmost 'i' it is 1 and for the rightmost 'i' it is 2. Since we know where each variable occurs in the association list, we can replace expressions of the form <'var' 'x'> with expressions of the form <'var' n> where n is the position of variable 'x' in the environment. This value n will be called the vp or variable position, as indicated in the new structure declaration for variables:

structure var: (vp)

Since we no longer use the variable names to look them up in the environment, we can eliminate them and represent the environment as a list of values rather than an association list. For instance the association list

< <'x' 2> <'y' 'Monterey'> <'z' <0 1 2>> >

will be simplified to

< 2 'Monterey' <0 1 2> >.

Then, looking up <'var' n> in environment 'a' is simply accomplished by a[n]. Since variables are no longer needed, the bound variable list can also be eliminated from the representation of abstractions. For instance 'λx{cons(x,<>)}' is represented by

<'lambda' <'prim' 'cons' <'var' 1> <'const' <> >> >.

The new structure definition for abstractions is:

structure lambda: (body)

This process of converting variables from names to numeric locations is one that is usually performed by a translator or compiler. Since we are translating by hand from the lambda calculus to its list representation, we must do this counting ourselves. It is generally true of programming language implementations that, as we have done here, the run-time efficiency of a program can be increased by doing more work at compile time.

There are only a few simple changes necessary to convert 'eval' to use the new numeric variables -- all simplifications. The rule for 'var's is simply altered to subscript the requred value out of the environment:

```
e|sif is-var then a[vp(e)]
```

The only other change is to the 'apply' procedure: since environments are simple lists of values the environment of evaluation is constructed by appending the operand to the front of the environment of definition.

```
and rec apply = proc( closure, actuals) begin
  let abs = ip(closure) in
    let env = append( actuals, ep(closure) ) in
      eval( body(abs), env)  end
```

To clarify these ideas we will trace the evaluation of the lambda expression:

λi{ λf{ λi{ f(i) }(2) }(λx{prod(i,x)}) }(3)

This is represented by the list:

```
E = <'call' I <<'const' 3>> >
I = <'lambda' <'call' F <X>> >
F = <'lambda' <'call' J <<'const' 2>> >>
J = <'lambda' <'call' <'var' 2> <<'var' 1>> >>
X = <'lambda' P>
P = <'prim' 'prod' <<'var' 2> <'var' 1>> >


Eval( E, <> ):
 apply( <'closure' I <>>, <3> ):
  env = append( <3>, <> ) = <3>
  eval( <'call' F <X>>, <3> ):
```

Evaluation proceeds much as before, until the stage when the application I(i) must be evaluated:

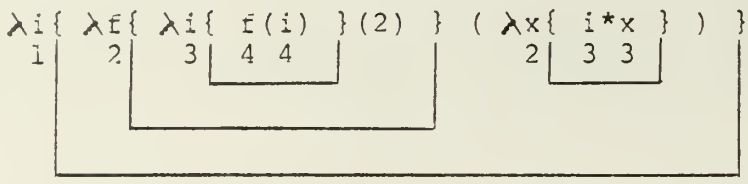
```
eval( <'call' <'var' 2> <<'var' 1>> >, <2 C 3> ):
 closure = eval(  <'var' 2>,   <2 C 3> ) = C
 actuals = evlis( <<'var' 1>>, <2 C 3> ) = <2>
 apply( C, <2> ):
  env = append( <2>, <3> ) = <2 3>
  eval( P, <2 3> ):
   actuals = evlis( <<'var' 2> <'var' 1>>, <2 3> )
           = <3 2>
   prim-apply( 'prod', <3 2> ) = 6
```

I.4.18 static nesting level. The approach used above is based on the replacement by the translator of variable names by numbers indicating the static distance of the use of the variable from its definition. This number will vary from one use of a variable to another. For instance, in

$$\lambda x\{\ cons(\ x,\ \lambda y\{\ cons(\ x,\ append(y,y)\ )\ \}(<A\ B>)\ )\ \}(C)$$

the first use of 'x' will be translated by <'var' 1> while the second use of 'x' will be translated by <'var' 2>. The translation process would be a little simpler if a single number was always associated with all uses of a given variable. We can do this by using the static nesting level of the definition of the variable as this number. The static nesting level of the

definition of a variable is one more than the number of scopes in which that definition of the variable is nested. In the diagram below, the static nesting levels of the variables are indicated:



The above diagram also indicates the static nesting level of the uses of variables. Notice that the static nesting level of the use of a variable is the sum of the static nesting level of its definition and the static distance to that use. Therefore we can calculate the static distance from the static nesting level or vice versa. This means that one way to alter 'eval' to use static nesting levels is to alter it to compute the static distance and then to extract the value from the environment in the previous way. There is a simpler way, however, since we can so construct the environment that the static nesting level can be used to directly access the environment. To understand the way this is done notice that with our present 'eval' the "innermost" variable is first in the environment and the "outermost" variable is last in the environment. To make most convenient use of static nesting levels we want to construct the environment so that the "outermost" variable (i.e. that at static nesting level 1) is the first (i.e. a[1]) and the "innermost" is last. This is accomplished by simply concatenating the latest variable values to the right of the environment rather than the left. The only required alteration to the interpreter is to switch 'actuals', and 'ep(closure)' in apply:

```
let rec apply = proc( closure, actuals) begin
  let abs = ip(closure) in
    let env = append( ep(closure), actuals ) in
      eval( body(abs), env)  end
```

The translation of lambda expressions into the list representation is simplified since a variable is always translated in the same way. The translation of our previous example is:

```
J = <'lambda' <'call' <'var' 2> <<'var' 3>> >>
P   <'prim' 'prod' <<'var' 1> <'var' 2>> >
```

EXERCISE: Trace the evaluation of this list.

I.4.19 multiple parameters. To keep the discussion simple, we have only been discussing the implementation of abstractions and applications with a single parameter. We will now extend our implementation to handle more than one parameter.

In our current implementation (using static nesting levels) the environment is represented by a list of the form

$$\langle v_1\ v_2\ \cdots\ v_n \rangle$$

where v_i is the value of the variable bound at static nesting level i. To handle multiple parameters we will simply replace each of these values with a list of the values bound at each static nesting level. That is, an environment has the form

$$\langle ar_1\ ar_2\ \cdots\ ar_n \rangle$$

where each ar_i is called the activation record (or call frame) for static nesting level i. An activation record is defined to be all of the information relevent to a call of a function. Each activation record has the form

$$\langle v_1\ v_2\ \cdots\ v_m \rangle$$

where the v_is are the values of the variables declared at the static nesting level corresponding to the activation record. To access a variable it is now necessary to use two coordinates, (ep,vp), where the environment part (ep) is the static nesting level of the activation record containing the variable, and the variable part (vp) is the position of the value of the variable within that activation record. For example, if we are in the environment

< <1 2 3> <'A' 'B' 'C'> <4 'D' <5 6> 'E'> >

then the (ep,vp) pair (2,1) will refer to 'A', the pair (3,2) will refer to 'D' and the pair (3,3) will refer to <5 6>. In particular, if 'a' is the environment, then 'a[ep][vp]' is the value of the variable corresponding to the pair (ep,vp). In order to make use of this new structure for the environment, it is necessary to translate variables into lists of the form

<'var' ep vp>

where (ep,vp) is the location of the variable. This corresponds to the structure:

structure var: (ep, vp)

Therefore, to handle multiple parameters, the rule for 'var's in

'eval' must be changed to:

```
elsif is-var(e) then ar[vp(e)]
  where ar = a[ep(e)]
```

The next issue to be addressed is the construction of these environments by the 'apply' procedure. If the environment of definition from the closure is

$$\langle ar_1 \cdots ar_n \rangle$$

and the actual parameters of the application are

$$\langle v_1\ v_2 \cdots v_m \rangle$$

then the environment in which the body of the abstraction is evaluated should be

$$\langle ar_1 \cdots ar_n \langle v_1\ v_2 \cdots v_m \rangle \rangle .$$

That is, a new activation record, $ar_{n+1}=\langle v_1 \cdots v_m \rangle$, is made the new last element in the environment. If 'ep' is the environment of definition and 'x' is the actual parameter list, then

```
consR( ep, x )
  where consR = proc(L,x){ append( L, cons(x,<>) ) }
```

will construct the environment of evaluation. The appropriately modified 'apply' is:

```
let rec apply = proc( closure, actuals) begin
  let abs = ip(closure) in
    let env = consR( ep(closure), actuals) in
      eval( body(abs), env )  end

and consR = proc(L,x){ append( L, cons(x,<>) ) }
```

The complete 'eval' is shown in the next figure. To demonstrate the operation of the new 'eval', we will trace the reduction of

$$\lambda xy\{\ \lambda z\{\ z*z + x\ \}\ (x+y)\ \}\ (2,3)$$

First we translate the lambda expression into our list representation, numbering the lambdas as before:

```
<call <lambda(1)
        <call <lambda(2) <prim sum <prim prod <var 2 1> <var 2 1
                                   <var 1 1>> >
              <prim sum <var 1 1> <var 1 2>> >>
      <const 2> <const 3>>
```

```
let rec Eval = proc(e,a) begin
  if    is-const(e)  then const-val(e)
  elsif is-var(e)    then ar[vp(e)]  where ar = a[ep(e)]
  elsif is-lambda(e) then closure( e, a)
  elsif is-prim(e)   then apply-prim( rator(e), actuals)
    where actuals = evlis( rands(e), a)
  elsif is-call(e)   then apply( closure, actuals)
    where closure = eval( rator(e), a)
    and   actuals = evlis( rands(e), a)
  elsif is-if(e)     then
    if eval( cond(e), a) then eval( t-branch(e), a)
    else eval( f-branch(e), a) endif
  endif end

and rec evlis  =  proc(L,a) begin
  if null(L) then <>
  else cons( eval(L[1],a), evlis(rest(L),a) )  endif  end

and rec apply = proc( closure, actuals) begin
  let abs = ip(closure) in
    let env = consR( ep(closure), actuals) in
      eval( body(abs), env)  end

and apply-prim = proc(f,x) begin
  if    f='first' then first(x[1])
  elsif f='cons'  then cons( x[1], x[2] )
  elsif ...
  endif end
in ...

and consR = proc(L,x){ append( L, cons(x,<>) ) }
```

Figure 5. Eval for Fixed-location Variables

Here we see that 'x', 'y' and 'z' have been translated as '<var 1 1>', '<var 1 2>' and '<var 2 1>', respectively, on the basis of their static nesting levels and their positions at that level, i.e. their (ep,vp) coordinates. The trace follows:

```
E = <'call' X <<'const' 2> <'const' 3>> >
X = <'lambda' <'call' Z <S>> >
Z = <'lambda' <'prim' 'sum' <P <'var' 1 1>> >>
S = <'prim' 'sum' <<'var' 1 1> <'var' 1 2>> >
P = <'prim' 'prod' <<'var' 2 1> <'var' 2 1>> >

Eval( E, <> ):
 apply( <'closure' X <> >, <2 3> ):
  env = consR( <>, <2 3> ) = <<2 3>>
  eval( <'call' Z <S>>, <<2 3>> ):
   actuals = evlis( <<'prim' 'sum'
                      <<'var' 1 1> <'var' 1 2>> >>, <<2 3>> )
           = < sum(2,3) > = <5>
    apply( <'closure' Z <<2 3>> >, <5> ):
     env = consR( <<2 3>>, <5> ) = <<2 3> <5>>
     eval( <'prim' 'sum' <P <'var' 1 1>> >, <<2 3> <5>> ):
      actuals = evlis( <P <'var' 1 1>>, <<2 3> <5>> )
              = < eval(P,<<2 3><5>>)  2 >
```

It is now necessary to compute the subexpression, eval(P,<<2 3><5>>).

```
                  eval( P, <<2 3> <5>> ):
                   actuals = evlis( <<'var' 2 1> <'var' 2 1>>,
                                    <<2 3> <5>> )
                           = <5 5>
                   prim-apply( 'prod', <5 5> ) = 25, hence,
        actuals = <25 2>
        prim-apply( 'sum', <25 2> ) = 27
```

You will recall that there were two possible versions of the single-parameter 'eval': one which used static nesting levels and one which used static distances. There are also two possible versions of the multiple-parameter version of 'eval': we can use either static nesting levels or static distances. When we investigate implementations of the lambda calculus further, later, we will find that there are circumstances when it is better to use the static distance and other circumstances when it is better to use the static nesting level. These are similar to implementation techniques called "static chains" and "displays", which are discussed later.

EXERCISE: Translate the lambda expression

$$\lambda xy\{\ \lambda z\{\ z*z + x\ \}\ (x+y)\ \}\ (2,3)$$

into the list representation using static distances instead of static nesting levels. Make the appropriate changes to 'eval' so

that it will work correctly with 'var's containing static distances and show that it works by tracing the evaluation of the above expression.

Chapter 5

I .5 Runtime Organization

I .5.1 Goals Defined

In this chapter we will study the stack implementation of the extended lambda calculus. That is, we will study how ELC can be translated into the instructions of conventional computers by using the data structure known as a stack. Since all block-structured languages, when stripped of their syntactic sugar, are essentially the lambda calculus, you will be learning how to implement these languages. (This important class of languages includes Algol, Pascal, PL/I and Ada.)

I .5.2 Stacks

The data structure used on most computers for implementing block-structured languages is the stack. Stacks derive their name from the push-down stacks that are often used for dispensing plates in cafeterias (figure 1).

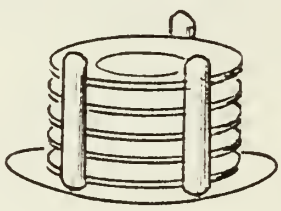


Figure 1. The Original Push-Down Stack

With these devices, each time a plate is removed from the top, the stack pops up to bring the next plate to the top. Conversely, whenever a plate is placed on the top of the stack, it pushes down the plates already there. Thus, a particular plate can be pushed onto the stack, and have other plates pushed on top of it, thereby hiding it. But if these plates are later popped off, then the plate we started with will again be at the top of the stack. This property, being able to save things (or information) by pushing them on a stack, makes the stack data structure particularly valuable in programming language implementation. Stacks are also know as push-down stores, deques and LIFOs (which means "last-in, first-out" and is pronounced "lie-foe"). To see the relevence of stacks to programming language implementation, we must next investigate postfix instructions.

I .5 .3 Postfix Instructions

We have seen how expressions can be written in functional form. For instance,

x + ab(y-z)

can be written

sum(x, prod(prod(a,b), dif(y,z)))

When an expression such as this is written without parentheses it is call prefix notation or Polish notation (after Jan Łukasiewicz, the Polish logician who invented it). The above expression written in prefix notation is:

+ x * * a b - y z

One of the important properties of Polish notation (and the reason it was invented) is that it is never necessary to use parentheses with it; it is often called parentheses free notation.

The reason Polish notation is also called prefix notation is that the operator is written before (pre-) its operands. E.g.

+ x y

The usual mathematical convention is called infix notation because the operator is written between (in-) its operands:

x + y

It should be obvious that if we wrote the operator after its operands, then we would be writing in postfix or reverse Polish notation. Reverse Polish notation is more important to computer scientists than Polish notation because reverse Polish can be evaluated easily by using a stack.

You may be familiar with "RPN" (or Reverse Polish Notation) calculators, such as those manufactured by Hewlitt-Packard and National Semiconductor. With these calculators expressions to be evaluated are entered in postfix notation and temporary results are held in a stack. For instance, to calculate

2 + 5 * 10,

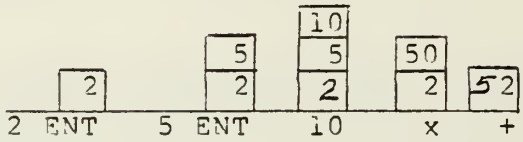
which is postfix notation is

2 5 10 * +,

we would hit the keys:

[2] [ENT] [5] [ENT] [1] [0] [x] [+]

The follwing diagram shows how the stack holds the intermediate results.



Notice how the stack holds the operands before the operation (e.g. 5 and 10) and the results after the operation (e.g. 50).

I .5 .4 The L-Machine Architecture

We will now investigate a computer with a stack architecture, i.e. a computer that uses a stack for the evaluation of postfix instructions. Although the L-Machine is not a real computer, it is essentially a simplification of several commercially available machines.

The L-Machine has three registers (high speed memory locations), called SP, EP and IP, and a memory addressed by consecutive natural numbers that can hold both data and instructions. The register names are mnemonic:

SP - Stack Pointer
EP - Environment Pointer
IP - Instruction Pointer

The SP register holds the address of the top of the stack, the EP register holds the address of the current activation record (explained further later) and the IP register holds the address of the next instruction to be executed. The L-Machine's storage architecture is summarized in the following figure.

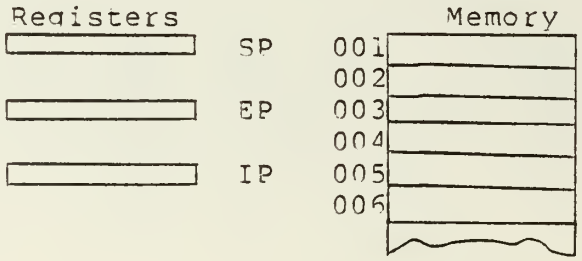
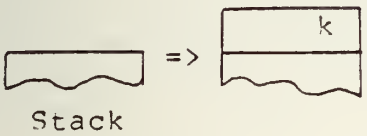


Figure 2. L-Machine Storage Architecture

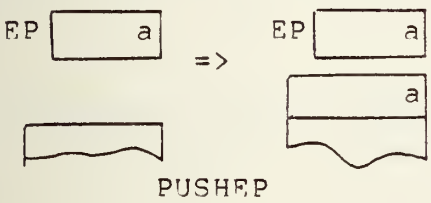
The L-Machine has some 25 instructions that it is able to execute. We will discuss each of these in the following paragraphs and indicate their operation with diagrams.

1 PUSH. The 'PUSH' operation pushes a constant value onto the stack. That is, if k is any constant (i.e., number, string, Boolean or list), then 'PUSH k' will increment the SP register and store k in the memory location addressed by SP. In diagramatic form:



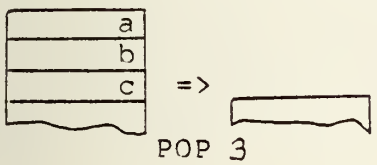
PUSH k

2 PUSHEP. The "PUSHEP" operation pushes the current contents of the EP register onto the stack:



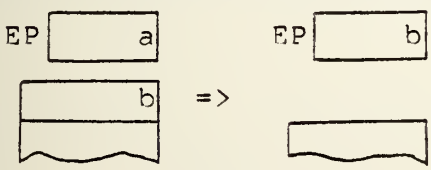
PUSHEP

3 POP. The 'POP' instruction discards elements from the top of the stack. That is, 'POP k' will pop the top k elements from the stack and discard them. E.g.,



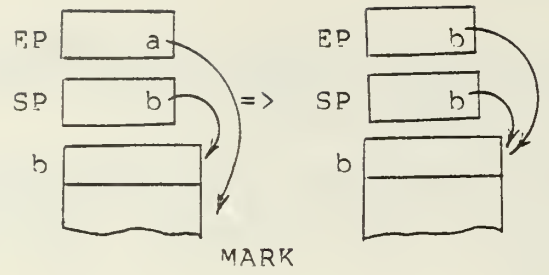
POP 3

4 SETEP. The 'SETEP' operation pops a value from the stack and places it in the EP register.



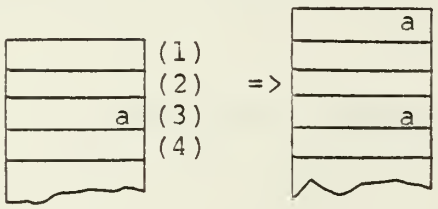
SETEP

5 MARK. The 'MARK' operation moves the current contents of the SP register into the EP register. The effect of this is to remember the location of the current top of the stack.



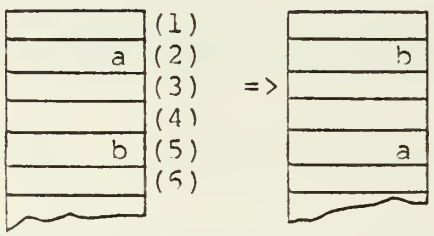
MARK

6 COPY. The 'COPY' operation copies a value from within a specified location of the stack, and places it on top of the stack. For instance, 'COPY 3' copies the value that is third from the top of the stack and pushes it:



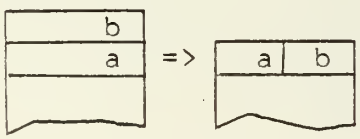
COPY 3

7 SWAP. The 'SWAP' operation exchanges two elements of the stack. For instance, 'SWAP 2,5' swaps the second and fifth elements from the top of the stack:



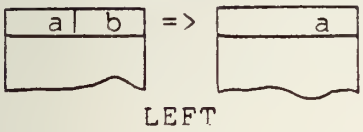
SWAP 2,5

8 PAIR. The 'PAIR' operation combines the top two stack elements into one; the two elements become the left and right halves of the resulting value. These two values can be extracted by 'LEFT' and 'RIGHT', described below.



PAIR

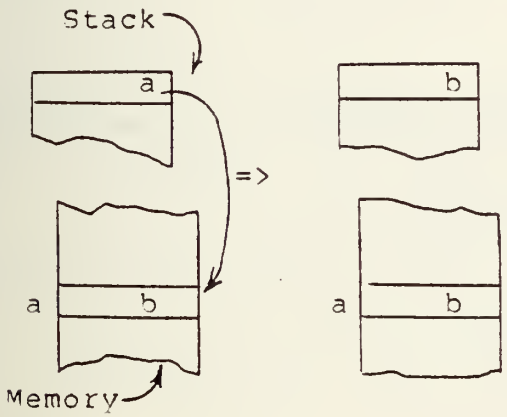
9 LEFT, RIGHT. The 'LEFT' operation extracts the left half of a value constructed by 'PAIR':



LEFT

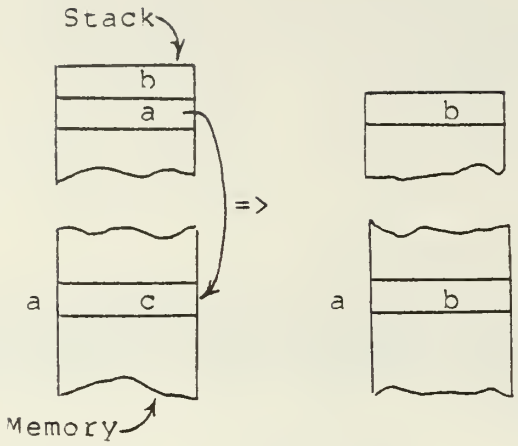
The 'RIGHT' operation extracts the right half.

10 VAL. The 'VAL' operation access the current contents of a location in memory. The 'VAL' operation takes an address from the top of the stack and replaces it with the contents of the memory location at that address.



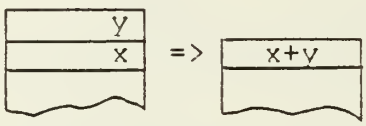
VAL

11 SET. The 'SET' operation changes the current contents of a memory location to a specified value. It takes a value and an address from the top of the stack and stores the value in the memory location with that address. The value is left on the top of the stack.



SET

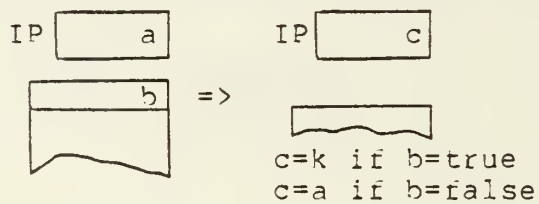
12 ADD, SUB, MUL, DIV. These operations perform addition, subtraction, multiplication and division. The operands are the two top elements of the stack, which are replaced by the results of the operation. This is essentially like an RPN calculator.



ADD
(SUB, MUL, DIV similar)

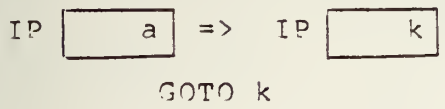
Any other primitive operations with which we wish to equip the L-Machine (such as the relationals, EQ, NE, GT, etc.) would operate analogously.

13 IF. The 'IF' operation performs a jump if the top of the stack is 'true'. In particular, the operation 'IF k' pops the stack and, if this value is 'true', transfers to the instruction at location k (by placing k in the IP register). If the top of the stack is false then execution continues at the instruction which follows the 'IF'.

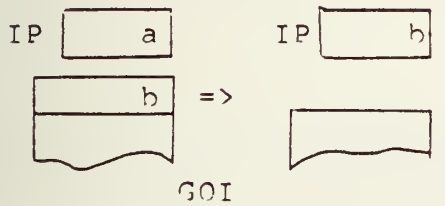


IF k

14 GOTO. The 'GOTO' operation transfers control to another location by placing its operand in the IP register.



15 GOI. The 'GOI' operation is an "indirect goto". That is, the top of the stack is a value that is used as the address of the next instruction to execute. This is accomplished by popping the stack into the IP register. GOI is used instead of GOTO when the destination address must be computed while the program is running.



I .5.5 Activation Record Structure As we learned in Chapter I.4 the activation record is the implementation mechanism that embodies the idea of the local environment of a computation. Activation records will also be important to our compiled implementation of the lambda calculus, so in this section we will study their implementation on a stack. You will remember that we defined an activation record to be all of the information relevent to a call of a procedure. In the Eval interpreter this was just the values of the bound variables of the function. We will see below that in the compiled implementation additional information must be included in the activation record.

Consider the following lambda expression, which we have already investigated several times:

```
let i = 3 in
  let f = proc(x){ x*i } in
    let i = 2 in
      f(i)
```

This can be represented by the contour diagram in figure 3. In this diagram we have drawn solid arrows from each contour to the statically enclosing contour. These arrows are called static links. A static chain is any contiguous sequence of static links, for instance from the i=2 contour to the f=λ{xi} contour to the i=3 contour. What the static chain provides is a search path for looking up variables. For instance, if we are executing in the environment indicated by the π in the above diagram, and wish to evaluate 'i', then we proceed as follows. First look in

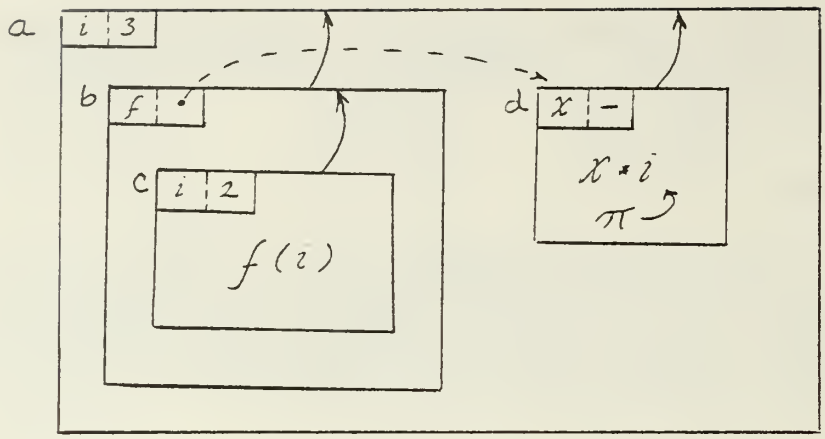


Figure 3. Contours

the local environment: is it defined here? No, only 'x' is defined here, so follow the static link to the enclosing environment: is it defined here? Yes, the value of 'i' is 3. If it hadn't been defined here we would have followed the static link and continued searching in the statically enclosing environment.

In our compiled implementation of the lambda calculus, activation records will be stored in a stack. We will use static links to find our way from one activation record to the next. For instance, when we begin executing 'f(i)' in contour (c), the stack's structure will be:

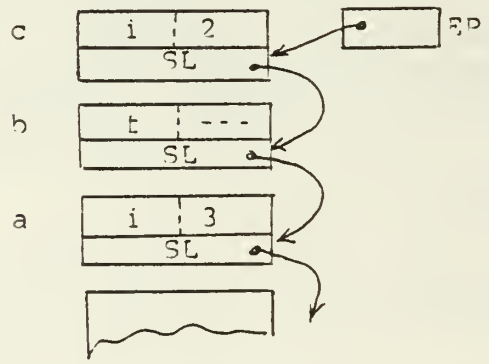


Figure 4. Before call to f

The activation records correspond exactly to the contours encountered in following the static chain from 'f(i)'. The EP register always points to the beginning of the static chain. Now, suppose we call the function bound to 'f'. This amounts to entering the contour (d) to execute 'x*i'. The activation record for 'f' will be pushed onto the top of the stack. Note, however that the static chain reflects the correct environment for 'x*i':

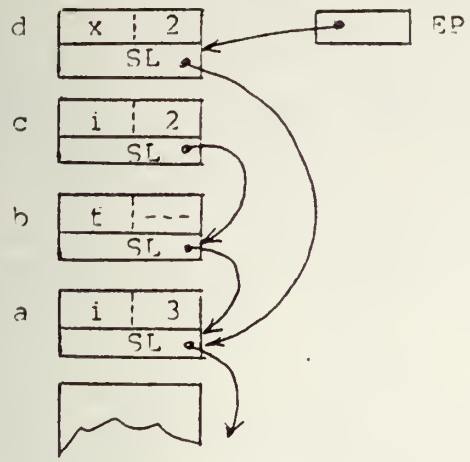


Figure 5. Inside f

In particular, when we come to evaluate 'i', we will follow the static chain to activation record (a), which binds 'i' to 3. When the function 'f' is exited the activation record (d) must be deleted from the stack and the situation restored to that of figure 4. To do this it is necessary to regain access to activation record (c), the activation record of the caller of 'f'. Doing this requires another link, the dynamic link, from each activation record to its caller. The sequence of dynamic links is called the dynamic chain. In figure 6 we see the same situation as in figure 5 (i.e. inside f) except that the dynamic chain is shown.

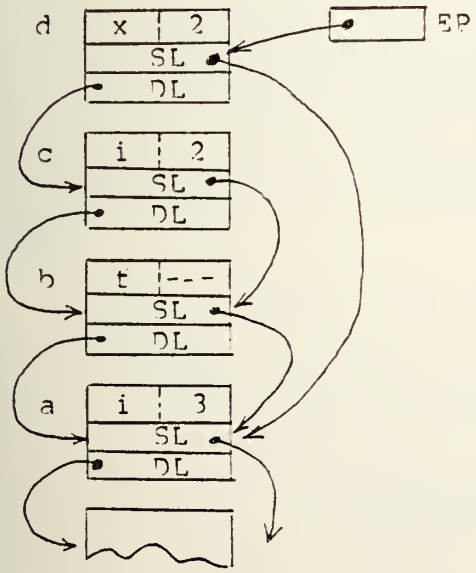


Figure 6. Inside f, with dynamic chain

Notice that the dynamic link always points to the next lower activation record in the stack.

So far we have seen that the activation record contains three pieces of information: the values of the bound variables, the static link and the dynamic link. There is one other type of information that is useful to include in the activation record: temporaries. When we discussed the stack evaluation of postfix expressions we saw that the stack holds operands until it is time to operate on them. We will use the stack for the same purpose here.

The actual format we will adopt for activation records is not the same as we have investigated so far. The differences are not important, however; we have merely reorganized the fields of the activation record to make it more convenient to access them with the instructions of the L-Machine. If we were compiling the lambda calculus for some other machine then some other arrangement might be better. The important thing is the information the activation record contains, irrespective of its arrangement:

* access to local environment (values of bindings)
* access to global environment (static link)
* access to caller (dynamic link)
* temporaries

This information must be in any activation record. The format we will use for the L-Machine is shown in figure 7.

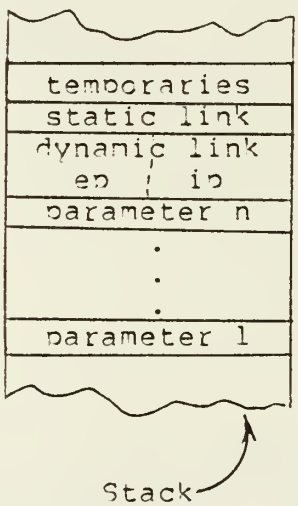


Figure 7. Activation Record Format

I .5 .6 Translation to L-Code

In this section we will discuss the L-Code (L-Machine instruction) translations for each construct in the extended lambda calculus. To aid this discussion we will use the notation $\wedge$[e] to denote the L-Code translation of the lambda expression e. For instance,

$\wedge$[25] = PUSH 25

The translations we will investigate in the rest of this section are not unique; different machines would suggest different code sequences; there are even different ways of accomplishing the same thing on the L-Machine. The important issue is not the particular instructions presented here, but rather the information flow required to execute the lambda calculus constructs. Since the lambda calculus forms a deep structure for most programming languages we are essentially studying the implementation methods for most programming languages.

1 Constants The translation of variables is the simplest, since all we have to do is to push the constant on the stack. For instance,

$\wedge$[25] = PUSH 25
$\wedge$['hat'] = PUSH 'hat'
$\wedge$[<1 <'hat' 2>>] = PUSH <1 <'hat' 2>>

or in general, for any constant c,

$\wedge$[c] = PUSH c

2 Variables In the discussion of the Eval interpreter we saw the use of (ep,vp) pairs to locate variables within the environment. We will do the same with our compiled implementation: the ep will locate the activation record holding the variable and the vp will locate the variable within that activation record. The static distance method of measuring the ep will be used.

First, consider how we get to the activation record in which the variable resides: if ep=1 then the variable resides in the current activation record, which is pointed to by the EP register. If ep=2 then the variable resides in the next most enclosing activation record, which we reach by following one link of the static chain form the current activation record. If ep=3 then we must follow two links of the static chain. In general, we must follow ep-1 links to get to the activation record containing variable (ep,vp).

Now let us consider the L-Code required to follow the static chain. Our goal will be to get the address of the activation

record ep onto the top of the stack. Consider the case ep=1. In this case the activation record is the current one, so

```
PUSHEP
```

will push its address onto the stack.

Now consider the case ep=2; here we must follow the static chain one link. Recall that both the EP register and the static links always point to the static link field of an activation record. See figure 8.

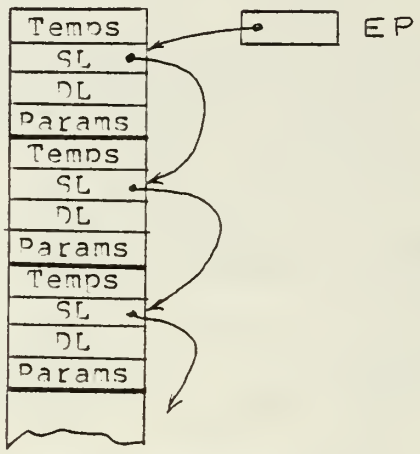


Figure 8. The Static Chain

Hence, after performing PUSHEP the top of the stack is the address of the static link for the current activation record. We can use VAL to load the contents of this location, i.e. to get the address of the next activation record in the static chain. Hence,

```
PUSHEP
VAL
```

will leave on the top of the stack the address of the ep=2 activation record.

Following the same reasoning we can see that if ep=3 then the address of the activation record is computed by:

```
PUSHEP
VAL
VAL
```

In general, the code to access the activation record at static distance ep is PUSHEP followed by ep-1 VALs. We will write this:

```
                PUSHEP
(ep-1) * { VAL
```

We have seen how to use the ep part of a variable's coordinates to get to the activation record in which it resides. We will now investigate the use of the vp part to locate the variable within that activation record. Figure 9 shows an activation record with several measurements indicated.

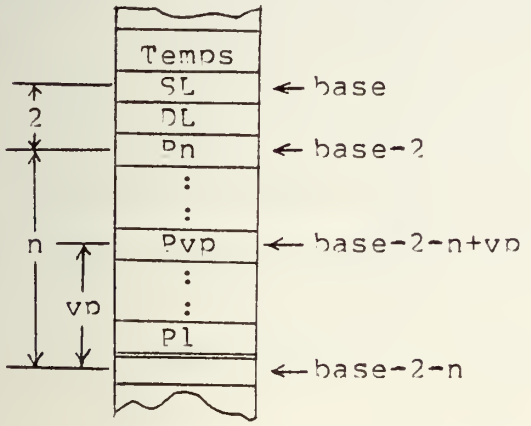


Figure 9. Activation Record Distances

If base is the base address of the activation record (i.e. the address of its static link) then

$$base-2-n+vp$$

is the address of the vp-th variable in that activation record. We can rewrite this as

$$base + (2+n-vp)$$

Notice that (2+n-vp) is a constant that depends only on the vp coordinate of the variable and the number of variables bound at lexical level ep. We will call this latter quantity n_{ep} and define

$$\delta(ep,vp) = 2+n_{ep}-vp$$

Thus the address of the variable with coordinates (ep,vp) is

$$base_{ep}+\delta(ep,vp)$$

It is very easy for a compiler to compute the quantity $\delta(ep,vp)$; it need only keep track of the number of variables declared at each lexical level.

Now we are ready to put together the two parts of the variable accessing mechanism. To get the address of variable (ep,vp) we need $base_{ep}$, the address of the activation record at static distance ep. This is exactly what is computed by:

```
                PUSHEP
(ep-1) * {      VAL
```

To get the address of the variable we must subtract δ(ep,vp):

```
                PUSH δ(ep,vp)
                SUB
```

Therefore the code for accessing the variable with coordinates (ep,vp) and leaving its value on the stack is:

```
Var ep,vp   =
                PUSHEP
(ep-1) * {      VAL
                PUSH δ(ep,vp)
                SUB
                VAL
```

We have given this sequence of instructions the "macro" name Var because it occurs so frequently. Macros like this are sometimes called code skeletons. For example, if the coordinates for 'x' are (3,2) and n_3=3 then the code for accessing 'x' is:

```
A[x]  =  Var 3,2  =
  PUSHEP
  VAL
  VAL
  PUSH 3
  SUB
  VAL
```

since $\delta(3,2) = 2+n_3-2 = 2+3-2 = 3$.

It must be emphasized again that the L-Code shown here is appropriate to the activation record format we have chosen. If the activation record format were different, then different instructions would be required. The important ideas to remember in accessing a variable with coordinates (ep,vp) are:

1) Find the activation record holding the variable by following the static chain for a static distance of ep.

2) Use the vp coordinate to access the variable within the activation record found in step (1).

We can now state the translation rule for a variable 'v' with coordinates (ep,vp):

$$\wedge[v] = \underline{Var}\ ep,vp$$

3 Applications

Recall that the steps involved in evaluating a application are:

1) Evaluate the operator.
2) Evaluate the operands.
3) Construct the environment of evaluation.
4) Evaluate the body of the function in this environment.

The code that we produce for a application will have to perform these same steps. It is:

$\wedge[\ f(e_1,\ldots,e_n)\ ]$ =
  $\wedge[f]$
  $\wedge[e_1]$
  .
  .
  .
  $\wedge[e_n]$
  Call n

The Call macro instruction will do the work of constructing an activation record for the new environment and executing the function in that environment (steps (3) and (4)).

EXAMPLE: Compile 'mod(7,2)'. Assume the coordinates of 'mod' are (3,1).

$\wedge[\ mod(7,2)\ ]$ =

```
  ∧[mod]
  ∧[7]
  ∧[2]
  Call 2  =

  Var 3,1
  PUSH 7
  PUSH 2
  Call 2
```

As noted above, the Call macro instruction is responsible for building the new activation record and for entering the procedure. Since the parameter values are already stacked, only the dynamic and static link parts of the new activation record remain

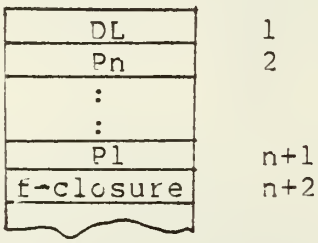
to be constructed. The steps are:

1) Construct dynamic link.
2) Construct static link.
3) Install the new current activation record.
4) Enter the function.

Since the dynamic link must preserve the state of the caller it has two parts: an ep to preserve the caller's environment and an ip to preserve the next instruction to execute. These can be saved by constructing a closure:

```
PUSHEP
PUSH r
PAIR
```

where r is the address of the next instruction following Call; this is where execution of the caller will resume when the called procedure returns. This leaves the stack in the state:



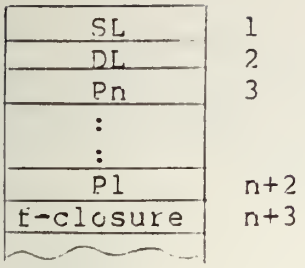
The static link must point to the environment of definition of the procedure. As the above diagram indicates, the (n+2) position from the top of the stack contains a closure for the procedure to be called (this resulted from evaluating the operator in step (1)). The static link is constructed by extracting the left half (ep) of this closure:

```
COPY n+2
LEFT
```

The third step, installing the new current activation record, is accomplished by placing the address of the new activation record into the EP register. Since the static link is currently at the top of the stack, this is accomplished by:

```
MARK
```

This leaves the stack in the state:



and completes construction of the activation record.

The fourth step is to enter the function. The address of the first instruction in the function is contained in the closure at position (n+3) in the stack. Thus transfer into the function is accomplished by:

```
COPY 3+n
RIGHT
GOI
```

Putting together the entire instruction sequence for Call yields:

```
Call n  =
  PUSHEP              |
  PUSH r              } build DL
  PAIR                |
  COPY n+2            } build
  LEFT                }   SL
  MARK                } install new AR
  COPY n+3            } get entry
  RIGHT               }   point
  GOI                 } enter the function
r:                    } the return location
```

It is to be emphasized again that the specific code sequence shown above is not so important as the general steps involved in a call:

1) Save the state of the call in the dynamic link.
2) Make the static link from the ep in the closure.
3) Enter the function at the location determined by the ip of the closure.

EXERCISE: Design the activation record format for a computer with which you are familiar. Write the sequence of instructions necessary to do a Call on this machine.

4 Abstractions

Recall that in the Eval interpreter the proper execution of an abstraction was ensured by packaging it together with its

environment into a closure. In the compiled case the ep part of a closure is a pointer to the activation record of definition and the ip part is the address of the first instruction of the code for the abstraction's body. This is constructed by:

```
PUSHEP
PUSH k
PAIR
```

where 'k' is the entry address of the function. The body of an abstraction '$\lambda v_1 \cdots v_n\{E\}$' is translated to:

```
k:
  ⋀[E]
  Return n
```

where Return is a "macro" instruction described below. Since we do not want the function body to be executed before it is called, we must jump over it. Therefore, the translation of an abstraction is:

$⋀[\lambda v_1 \cdots v_n\{E\}]$ =

```
  PUSHEP   ep  |
  PUSH k   ip  } build closure
  PAIR         |
  GOTO s       } skip body
k:⋀[E]
  Return n
s:
```

The Return instruction must accomplish the following tasks:

1) Pass the returned-value from the callee to the caller.
2) Restore the state of the caller from the dynamic link.
3) Delete the callee's activation record.

The state of the stack before the Return is:

answer	1
SL	2
DL	3
Pn	4
:	
:	
P1	n+3
f-closure	n+4

The returned-value ('answer' in the above diagram) can be placed above the caller's temporaries and two words of the activation

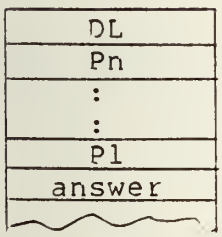
record deleted by:

```
SWAP 1,n+4
POP  2
```

The environment of the caller is restored by:

```
COPY 1
LEFT
SETEP
```

leaving the stack in the form:



The address to which to return to the caller is extracted from the dynamic link and saved above the answer in the stack by:

```
RIGHT
SWAP 1,n+1
POP  n
```

The 'POP n' instruction deletes the n parameter values, exposing the return-address for a GOI back to the caller. The resulting code is:

```
Return n  =
  SWAP 1,n+4        } save answer
  POP  2            } delete closure, SL
  COPY 1            |
  LEFT              } restore EP from DL
  SETEP             |
  RIGHT             } get return address from DL
  SWAP 1,n+1        } save return address
  POP  n            } delete parameters
  GOI               } reenter caller
```

5 Conditionals

The compiled L-Code for handling conditionals will operate similarly to the Eval interpreter. That is, for

[B -> T | F]

we must first evaluate B and if it is true evaluate T, otherwise

evaluate F. The translation to accomplish this is:

```
⋀[ [ B -> T | F ] ]  =

    ⋀[B]
    IF t
    ⋀[F]
    GOTO x
t:  ⋀[T]
x:
```

6 Blocks

In section I.4.20 we saw that the efficiency of blocks (local declarations) could be improved by implementing them in Eval directly, rather than as procedure invocations. The same is true with respect to the L-Code implementation of blocks, since a simpler activation record structure will suffice and since it will not be necessary to construct or decompose closures. Indeed, since a block is always invoked in its environment of definition, its static and dynamic links are always the same. Further, since a block is always invoked from the same place, it is not necessary to save the IP of the caller. Although this means that these items could be omitted entirely from the activation record for blocks, we will include space for them so that the format will agree with that of a procedure's activation record. This will allow us to use the translation of variables already discussed regardless of whether the variable was bound by a procedure or a block. In an actual compiler we might choose to use two different formats at the expense of a more complicated translation process.

With this explanation done we can proceed with the translation of blocks. This makes use of two "macro" instructions which create and delete the block's activation record:

⋀[let v_1=e_1 and ··· and v_n=e_n in B] =

⋀[e_1]
  .
  .
  .
⋀[e_n]
Begin
⋀[B]
End n

The Begin and End instructions are simplifications of Call and Return:

Begin =

```
  PUSHEP      } ep          } build DL

  PUSHEP      } SL
  MARK        } install new current AR
```

End n =

```
  SWAP 1,3+n  } save answer
  POP  2      } delete first local and SL
              } restore EP
  SETEP       }   from DL
  POP  n-1    } discard rest of locals
```

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93940	2
Office of Research Administration Code 012A Naval Postgraduate School Monterey, CA 93940	1
Chairman, Code 52Bz Department of Computer Science Naval Postgraduate School Monterey, CA 93940	40
Professor Bruce J. MacLennan, Code 52Ml Department of Computer Science Naval Postgraduate School Monterey, CA 93940	20